Java SE 8

Описана Java SE 8

Базовый курс

Кей С. Хорстманн

Java SE 8

Базовый курс

Core Java

for the Impatient

Cay S. Horstmann

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Java SE 8

Базовый курс

Кей С. Хорстманн

Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2015

ББК 32.973.26-018.2.75
Х82
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Хорстманн, Кей С.

Х82 Java SE 8. Базовый курс. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 464 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2004-1 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание
Кей С. Хорстманн
Java SE 8. Базовый курс

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 27.08.2015. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 37,41. Уч.-изд. л. 24,4.
Тираж 300 экз. Заказ № 4679.

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2004-1 (рус.)
ISBN 978-0-321-99632-9 (англ.)

Оглавление

Содержание

Посвящение

Чи — самому терпеливому существу на свете.

Об авторе

Кей С. Хорстманн — автор книг *Scala for the Impatient* (издательство Addison-Wesley, 2012 г.) и *Java SE 8 for the Really Impatient* (издательство Addison-Wesley, 2014 г.; в русском переводе книга вышла под названием *Java SE 8. Вводный курс* в ИД "Вильямс", 2014 г.), а также основной автор двухтомного издания *Core Java™, Volumes I and II, Ninth Edition* (издательство Prentice Hall, 2013 г.; в русском переводе это издание вышло в двух томах под общим названием *Java. Библиотека профессионала* в ИД "Вильямс", 2014 г.). Он также написал десяток других книг для профессиональных программистов и изучающих вычислительную технику. Кей служит профессором на кафедре вычислительной техники при университете штата Калифорния в г. Сан-Хосе и носит почетное звание "Чемпион по Java".

Предисловие

Язык программирования Java существует уже около двадцати лет. В классическом издании *Core Java*™ (в русском переводе это девятое издание вышло в двух томах под общим названием *Java. Библиотека профессионала* в ИД "Вильямс", 2014 г.) во всех подробностях описывается не только сам язык, но и все основные его библиотеки, а также многочисленные изменения в очередных версиях. Все это занимает два тома и около 2 тысяч страниц текста. Но версия Java 8 совершенно меняет дело. Многие из прежних принципов и приемов программирования на Java больше не нужны, а изучить Java можно много быстрее и проще. В этой книге раскрываются наилучшие стороны современной версии Java, чтобы их можно было быстро применить на практике.

Как и в предыдущих моих книгах, написанных в стиле изложения "для нетерпеливых", мне хотелось сразу ввести вас, читатель, в курс дела, не вдаваясь в подробное рассмотрение преимуществ одних понятий над другими, чтобы показать, что именно вам нужно знать для решения конкретных задач программирования. Поэтому материал в этой книге представлен небольшими фрагментами, организованными таким образом, чтобы вы могли быстро извлечь полезные сведения по мере надобности.

Если у вас имеется опыт программирования на других языках, например C++, JavaScript, Objective C, PHP или Ruby, то эта книга поможет вам грамотно программировать и на Java. В ней рассматриваются все особенности Java, которые должны быть известны разработчику, включая эффективные лямбда-выражения и потоки данных, внедренные в версии Java 8. В этой книге делаются ссылки на устаревшие понятия и языковые конструкции, которые могут встретиться в унаследованном коде, но я не буду останавливаться на них подробно.

Главная причина использования Java состоит в том, чтобы решать задачи параллельного программирования. Имея в своем распоряжении параллельные алгоритмы и потокобезопасные структуры данных, доступные в готовом виде в библиотеке Java, прикладные программисты должны совсем иначе отнестись к параллельному программированию. В этой книге под совершенно новым углом зрения будет показано, как пользоваться эффективными библиотечными средствами вместо чреватых ошибками низкоуровневых конструкций.

В традиционных книгах по Java основное внимание уделяется программированию пользовательского интерфейса, но ныне лишь немногие разработчики создают пользовательские интерфейсы для настольных систем. Если вы предполагаете пользоваться Java для программирования на стороне сервера или на платформе Android, данная книга поможет вам делать это эффективно, не отвлекаясь на код графического пользовательского интерфейса настольных систем.

И наконец, книга написана для прикладных, но не для системных программистов или тех, кто изучает программирование в учебных заведениях. В этой книге рассматриваются вопросы, которые приходится решать прикладным программистам,

включая протоколирование и обработку файлов. Но из нее нельзя узнать, как реализовать связный список вручную или написать код для веб-сервера.

Надеюсь, что вам понравится это беглое введение в современную версию Java, а также на то, что оно сделает вашу работу с Java более производительной и приятной. Если вы обнаружите ошибки в книге или имеете дельные предложения по ее усовершенствованию, оставьте комментарии на веб-странице, доступной по адресу `http://horstmann.com/javaimpatient`. На данной странице вы найдете также ссылку на архивный файл, содержащий весь код из примеров, приведенных в книге.

Благодарности

Как всегда, выражаю искреннюю благодарность своему редактору Грегу Дёнчу (Greg Doench), которому принадлежит замысел издать краткое пособие для прикладных программистов, чтобы помочь им поскорее освоить версию Java 8. Как и прежде, Дмитрий и Алина Кирсановы превратили рукопись формата XHTML в привлекательную книгу, сделав это поразительно быстро и в то же время уделив должное внимание деталям. Благодарю также рецензентов книги, обнаруживших в ней немало досадных ошибок и внесших ряд дельных предложений по ее улучшению. Среди них хотелось бы отметить Андреса Алмирая (Andres Almiray), Брайна Гётца (Brian Goetz), Марти Холла (Marty Hall), Марка Лоуренса (Mark Lawrence), Дуга Ли (Doug Lea), Саймона Риттера (Simon Ritter), Йошики Шибата (Yoshiki Shibata) и Кристиана Улленбоома (Christian Ullenboom).

Кей Хорстманн,
Биль/Бьенн, Швейцария,
январь 2015 г.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д.43, стр. 1

в Украине: 03150, Киев, а/я 152

Глава 1

Основополагающие структуры программирования

В этой главе...

Из этой главы вы узнаете об основных типах данных и управляющих структурах языка Java. При этом предполагается, что у вас имеется достаточный опыт программирования на других языках и что вы знакомы с такими понятиями, как переменные, циклы, вызовы функций и массивы, хотя и в другом синтаксисе. Материал этой главы послужит толчком к ускоренному изучению Java. В ней даются некоторые рекомендации относительно применения на практике наиболее полезных составляющих прикладного программного интерфейса Java API для манипулирования типами данных из общей системы типов.

Основные положения этой главы приведено ниже.

1. Все методы в Java объявляются в классе. Нестатический метод вызывается для объекта того класса, к которому этот метод принадлежит.
2. Статические методы не вызываются для объектов. Выполнение программы начинается со статического метода `main()`.
3. В языке Java имеется восемь примитивных типов данных: пять целых типов, два числовых с плавающей точкой и один логический тип `boolean`.
4. Операции и управляющие структуры в Java очень похожи на аналогичные языковые средства C или JavaScript.
5. Общие математические функции предоставляются в классе `Math`.
6. Объекты типа `String` являются последовательностями символов, а точнее — кодовыми точками Юникода в кодировке UTF-16.
7. С помощью объекта `System.out`, представляющего стандартный поток вывода, можно выводить результаты для отображения в окне терминала. Класс `Scanner` тесно связан с объектом `System.in`, представляющим стандартный поток ввода, и позволяет читать данные, вводимые с терминала.
8. Массивы и коллекции могут использоваться для накопления элементов одного и того же типа.

1.1. Первая программа на Java

Изучение любого нового языка программирования по традиции начинается с программы, выводящей сообщение `"Hello, World!"` (Здравствуй, мир!). Именно этим мы и займемся в последующих разделах.

1.1.1. Анализ программы "Hello, World!"

Итак, приведем без лишних слов исходный код программы `"Hello, World!"` на Java.

```
package ch01.sec01;

// Первая программа на Java

public class HelloWorld {
  public static void main(String[] args) {
```

```
    System.out.println("Hello, World!");
  }
}
```

Ниже приведен краткий анализ этой программы.

- Язык Java является объектно-ориентированным. В программе на Java приходится манипулировать *объектами*, чтобы выполнить с их помощью нужные действия. Практически каждый манипулируемый объект принадлежит к определенному *классу*. В классе определяется назначение объекта. В языке Java весь код определяется в классе. Более подробно объекты и классы будут рассмотрены в главе 2. А рассматриваемая здесь программа состоит из единственного класса `HelloWorld`.
- *Метод* `main()` представляет собой функцию, объявляемую в классе. Это первый метод, вызываемый при выполнении программы. Он объявляется как статический (`static`), а это означает, что данный метод не оперирует никакими объектами. (Когда метод `main()` вызывается, имеется совсем немного предопределенных объектов, но ни один из них не является экземпляром класса `HelloWorld`.) Данный метод объявляется также как пустой (`void`), а это означает, что он не возвращает никакого значения. Подробнее о назначении объявления параметров `String[] args` см. далее, в разделе 1.8.8.
- Многие языковые средства Java можно объявлять как открытые (`public`) или закрытые (`private`), хотя имеются и другие уровни их доступности. В данном примере класс `HelloWorld` и метод `main()` объявляются как `public`, что наиболее характерно для классов и методов.
- *Пакет* состоит из взаимосвязанных классов. Каждый класс целесообразно размещать в пакете, чтобы различать несколько классов с одинаковым именем. В данной книге в качестве имен пакетов используется нумерация глав и разделов. Таким образом, полное имя рассматриваемого здесь класса будет следующим: `ch01.sec01.HelloWorld`. Подробнее о пакетах и их условных обозначениях речь пойдет в главе 2.
- Строка кода, начинающаяся со знаков `//`, содержит комментарий. Все символы от знаков `//` и до конца строки игнорируются компилятором и предназначены только для чтения человеком, а не машиной.
- И наконец, рассмотрим тело метода `main()`. Оно состоит из единственной строки кода, содержащей команду для вывода сообщения в объект `System.out`, представляющий стандартный поток вывода результатов выполнения программ на Java.

Как видите, Java не является языком написания сценариев, на котором можно быстро написать несколько команд. Этот язык специально предназначен для разработки крупных программ, которые выгодно отличаются организацией в классы и пакеты.

Кроме того, язык Java довольно прост и единообразен. В некоторых языках допускаются глобальные переменные и функции, а также переменные и методы в классах. А в Java все объявляется в классе. Подобное единообразие может привести

к несколько многословному коду, но в то же время оно упрощает понимание назначения программы.

НА ЗАМЕТКУ. Как пояснялось выше, комментарий простирается от знаков // и до конца строки. Можно составить и многострочные комментарии, ограничив их разделителями /* и */, как показано ниже.

```
/*
   Это первый пример программы на Java в данной книге.
   Эта программа выводит традиционное приветствие "Hello, World!".
*/
```

Имеется и третий стиль комментария, называемого *документирующим*. В этом случае комментарий ограничивается разделителями /** и */, как будет *показано в следующей главе*.

1.1.2. Компилирование и выполнение первой программы на Java

Чтобы скомпилировать и выполнить рассматриваемую здесь программу на Java, нужно установить комплект разработки программ на Java под названием Java Development Kit (JDK), а дополнительно, хотя и не обязательно, — интегрированную среду разработки (ИСР). Кроме того, следует загрузить исходный код из данного примера, обратившись на сопровождающий данную книгу веб-сайт по адресу `http://horstmann.com/javaimpatient`. Читать инструкции по установке программного обеспечения неинтересно — их можно найти на том же веб-сайте.

Установив комплект JDK, откройте окно терминала, перейдите к каталогу, содержащему каталог `ch01`, и выполните приведенные ниже команды. Уже знакомое вам приветствие появится в окне терминала (рис. 1.1).

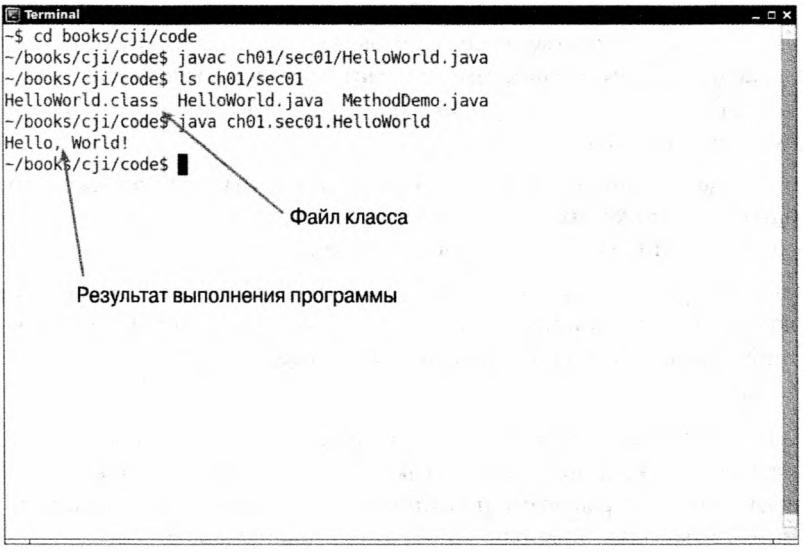


Рис. 1.1. Выполнение первой программы на Java в окне терминала

```
javac ch01/sec01/HelloWorld.java
java ch01.sec01.HelloWorld
```

Обратите внимание на двухэтапный процесс выполнения программы. Сначала по команде **javac** исходный код Java *компилируется* в промежуточное представление, называемое *байт-кодом*, а результат сохраняется в *файлах классов*. Затем по команде **java** запускается *виртуальная машина*, загружающая файлы классов и выполняющая байт-коды.

После компиляции байт-коды можно выполнять на любой виртуальной машине Java, будь то настольный компьютер или вычислительное устройство в далекой галактике. Обещание "написано однажды — выполняется везде" было важным критерием разработки языка Java.

НА ЗАМЕТКУ. Компилятор **javac** вызывается с именем *файла*, имеющего расширение **.java**, причем знаками косой черты разделяются сегменты пути к этому файлу. Загрузчик виртуальной машины **java** вызывается с именем *класса*, причем знаками точки разделяются сегменты пакета, а расширение файла не указывается.

Чтобы выполнить рассматриваемую здесь программу в ИСР, нужно сначала создать проект, как поясняется в инструкциях по установке, а затем выбрать класс `HelloWorld` и дать ИСР команду на выполнение программы. На рис. 1.2 показано, как

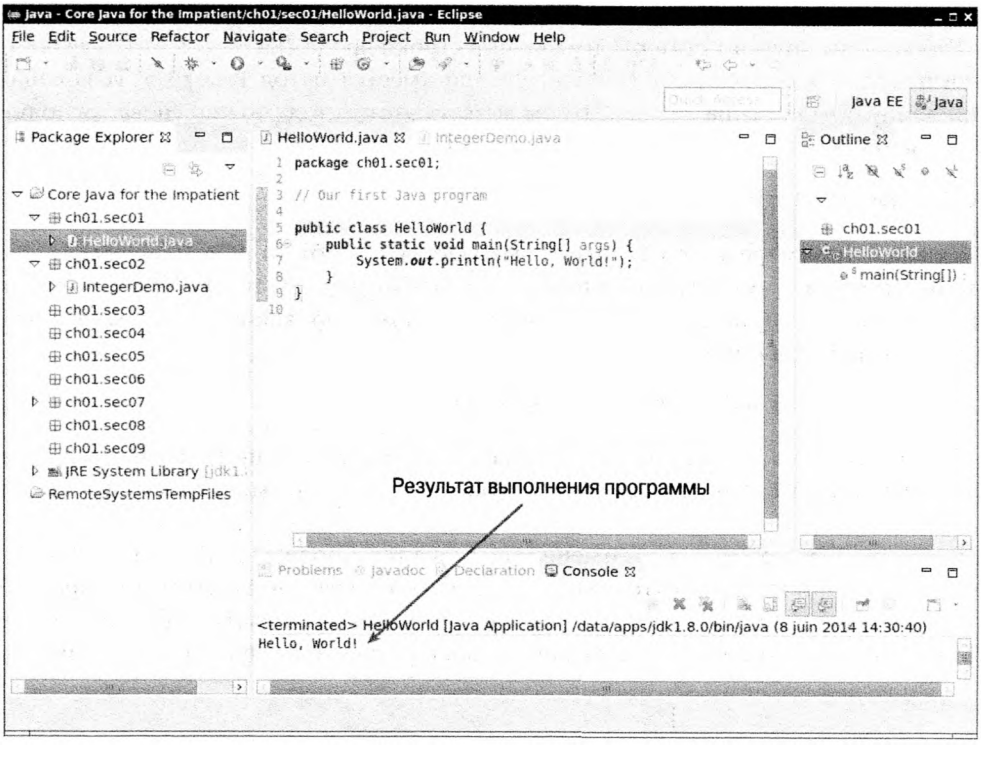


Рис. 1.2. Выполнение первой программы на Java в ИСР Eclipse

это выглядит в Eclipse — распространенной ИСР, хотя имеются и другие отличные ИСР. Освоившись немного с программированием на Java, опробуйте несколько ИСР, чтобы выбрать для себя наиболее подходящую.

Примите поздравления! Вы только что выполнили свою первую программу на Java, следуя освященному временем ритуалу вывода сообщения "Hello, World!". А теперь можно перейти к рассмотрению основ языка Java.

1.1.3. Вызовы методов

Рассмотрим более подробно следующий оператор, оказывающийся единственным в теле метода main():

```
System.out.println("Hello, World!");
```

В приведенном выше операторе System.out является объектом. Это *экземпляр* класса PrintStream. В классе PrintStream имеются методы println(), print() и т.д., которые называются *методами экземпляра*, поскольку они оперируют объектами или экземплярами класса. Чтобы вызвать метод экземпляра для объекта, следует воспользоваться *записью через точку*, как показано ниже.

```
объект.ИмяМетода(аргументы)
```

В рассматриваемом здесь примере указан единственный аргумент метода — символьная строка "Hello, World!".

Рассмотрим другой пример. Символьные строки вроде "Hello, World!" являются экземплярами класса String. В классе String имеется метод length(), возвращающий длину объекта типа String. Чтобы вызвать этот метод, можно снова воспользоваться записью через точку, как показано ниже.

```
"Hello, World!".length()
```

В данном примере метод length() вызывается для объекта "Hello, World!", и у него отсутствуют аргументы. В отличие от метода println(), метод length() возвращает результат. В приведенном ниже примере показано, как воспользоваться этим результатом, чтобы вывести его.

```
System.out.println("Hello, World!".length());
```

Опробуйте этот пример. Создайте для этого программу на Java с приведенным выше оператором и выполните ее, чтобы выяснить длину символьной строки "Hello, World!".

В отличие от готовых к применению объектов вроде System.out и "Hello, World!", большинство объектов в Java приходится *строить*. Рассмотрим еще один простой пример. Объект класса Random способен генерировать случайные числа. А строится объект типа Random с помощью оператора new следующим образом:

```
new Random()
```

После имени класса в этом операторе указывается список аргументов для построения объекта. В данном примере этот список пуст. Для построенного таким образом объекта можно вызвать метод. Так, в результате вызова

```
new Random().nextInt()
```

получается следующее целое число, предоставляемое вновь построенным генератором случайных чисел.

Если для объекта требуется вызвать несколько методов, его следует сохранить в переменной (подробнее об этом — далее, в разделе 1.3). В следующем примере кода выводятся два случайных числа:

```
Random generator = new Random();
System.out.println(generator.nextInt());
System.out.println(generator.nextInt());
```

НА ЗАМЕТКУ. Класс **Random** объявляется в пакете **java.util**. Чтобы воспользоваться им в программе, достаточно ввести оператор **import** в ее исходный код следующим образом:

```
package ch01.sec01;

import java.util.Random;

public class MethodDemo {
   ...
}
```

Более подробно пакеты и оператор **import** будут рассматриваться в главе 2.

1.2. Примитивные типы

Простейшие типы данных в Java называются *примитивными*. Четыре из них являются целочисленными; два — числовыми с плавающей точкой; один — символьным типом char, служащим для кодирования символьных строк; и еще один — логическим типом boolean для обозначения логических значений "истина" и "ложь". Все эти типы данных рассматриваются в последующих разделах.

1.2.1. Целочисленные типы

Целочисленные типы служат для обозначения целых чисел без дробной части. Допускаются отрицательные целочисленные значения. В языке Java предоставляются четыре целочисленных типа (табл. 1.1).

Таблица 1.1. Целочисленные типы в Java

Тип	Требования к хранению	Допустимый диапазон значений (включительно)
int	**4** байта	От **-2147483648** до **2147483647** (т.е. свыше 2 млрд)
long	**8** байт	От **-9223372036854775808** до **9223372036854775807**

Окончание табл. 1.1

Тип	Требования к хранению	Допустимый диапазон значений (включительно)
`short`	2 байта	От -32768 до 32767
`byte`	1 байт	От -128 до 127

НА ЗАМЕТКУ. Константы `Integer.MIN_VALUE` и `Integer.MAX_VALUE` обозначают наименьшее и наибольшее целые значения соответственно. В классах `Long`, `Short` и `Byte` также имеются константы `MIN_VALUE` и `MAX_VALUE`.

На практике чаще всего применяется тип `int`. Если требуется представить количество обитателей планеты, то следует прибегнуть к типу `long`. А типы `byte` и `short` предназначены для такого специального применения, как низкоуровневая обработка файлов или экономное размещение крупных массивов в оперативной памяти.

НА ЗАМЕТКУ. Если типа `long` оказывается недостаточно, следует воспользоваться классом `BigInteger`. Подробнее в этом — в разделе 1.4.6.

Допустимые диапазоны значений целочисленных типов в Java не зависят от машины, на которой будет выполняться программа. Ведь язык Java разработан по принципу "написано однажды — выполняется везде". Напротив, в программах на C и C++ целочисленные типы зависят от процессора, для которого компилируется программа.

Целочисленные литералы типа `long` обозначаются с суффиксом `L` (например, `4000000000L`). А для типов `byte` и `short` соответствующий синтаксис литералов не предусмотрен. Поэтому для обозначения констант этих типов приходится пользоваться приведением типов (например, `(byte) 127`). Подробнее о приведении типов — в разделе 1.4.4.

Шестнадцатеричные литералы обозначаются с префиксом `0x` (например, `0xCAFEBABE`), а двоичные значения — с префиксом `0b`. Например, двоичное значение `0b1001` равно десятичному значению `9`.

ВНИМАНИЕ. Восьмеричные значения обозначаются с префиксом `0`. Например, восьмеричное значение `011` равно десятичному значению `9`. Во избежание путаницы лучше отказаться от применения восьмеричных литералов и начальных нулей.

В числовые литералы можно вводить знаки подчеркивания, например, `1_000_000` или `0b1111_0100_0010_0100_0000` для обозначения миллиона. Это делается только ради повышения удобочитаемости подобных литералов, а компилятор просто удаляет из них знаки подчеркивания.

НА ЗАМЕТКУ. Целочисленные типы в Java имеют знак. Но если работать с целыми значениями, которые всегда положительны и действительно требуют дополнительного бита, то можно воспользоваться методами, интерпретирующими подобные значения как не имеющие знака. Например, вместо диапазона целых значений от `-128` до `127` можно выбрать диапазон значений от `0` до `255`

и представить его значением b типа byte. Такие операции, как сложение и вычитание, окажутся вполне работоспособными в силу характера двоичной арифметики. А для выполнения остальных операций следует вызвать метод Byte.toUnsignedInt(b) и получить значение типа int в пределах от 0 до 255.

1.2.2. Числовые типы с плавающей точкой

Числовые типы с плавающей точкой обозначают числа с дробной частью. Оба числовых типа с плавающей точкой приведены в табл. 1.2.

Таблица 1.2. Числовые типы с плавающей точкой в Java

Тип	Требования к хранению	Допустимый диапазон значений (включительно)
float	4 байта	Приблизительно ±3.40282347E+38F (с точностью до 6-7 значащих десятичных цифр)
double	8 байт	Приблизительно ±1.79769313486231570E+308 (с точностью до 15 значащих десятичных цифр)

Много лет назад, когда оперативная память была скудным ресурсом, часто употреблялись четырехбайтовые числа с плавающей точкой. Но их точности до семи десятичных цифр явно не хватает, и поэтому сейчас по умолчанию употребляются восьмибайтовые числа с плавающей точкой, имеющие "двойную" точность. А типом float имеет смысл пользоваться только в том случае, если требуется хранить большие массивы чисел с плавающей точкой.

НА ЗАМЕТКУ. Числовые литералы с плавающей точкой можно обозначать и в шестнадцатеричной форме. Например, десятичное число 0.0009765625 = 2^{-10} может быть обозначено в шестнадцатеричной форме как 0x1.0p-10. Для обозначения показателя степени в шестнадцатеричной форме употребляется буква p, а не e, поскольку буква e обозначает шестнадцатеричную цифру. Обратите внимание на то, что показатель степени указывается в десятичной форме, т.е. в степени два, тогда как цифры — в шестнадцатеричной форме.

Для обозначения положительной и отрицательной бесконечности, а также величины "не число" в форме с плавающей точкой служат специальные константы Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY и Double.NaN соответственно. Например, в результате вычисления выражения 1.0 / 0.0 получается положительная бесконечность, а в результате вычисления выражения 0.0 / 0.0 или извлечения квадратного корня из отрицательного числа — значение NaN (не число).

ВНИМАНИЕ. Все значения "не число" считаются отличающимися друг от друга. Следовательно, с помощью выражения if (x == Double.NaN) нельзя проверить, содержит ли переменная x значение NaN. Вместо этого следует употребить проверочное выражение if (Double.isNaN(x)), вызвав в нем метод isNaN(). В классе Double имеются также методы isInfinite() и isFinite() для проверки числовых значений с плавающей точкой на положительную и отрицательную бесконечность и на бесконечность и не число соответственно.

Числа с плавающей точкой непригодны для финансовых расчетов, где ошибки округления недопустимы. Например, оператор `System.out.println(2.0 - 1.1)` выводит результат `0,8999999999999999`, а не `0,9`, как следовало бы ожидать. Подобные ошибки округления обусловлены тем, что числа с плавающей точкой представлены в двоичной системе счисления. Точного представления дроби `1/10` в двоичной системе счисления не существует, как, впрочем, и точного представления дроби `1/3` в десятичной системе счисления. Если требуются точные расчеты без ошибок округления, следует воспользоваться классом `BigDecimal`, представленным далее, в разделе 1.4.6.

1.2.3. Тип `char`

Тип `char` обозначает "кодовые единицы" в кодировке символов UTF-16, употребляемой в Java. Подробнее об этом — ниже, в разделе 1.5. Вряд ли вам придется часто пользоваться типом `char`.

Иногда в исходном коде программ на Java встречаются символьные литералы, заключенные в одиночные кавычки. Например, символьный литерал `'J'` имеет десятичное значение **74** (или шестнадцатеричное значение **4A**) и представляет кодовую единицу, обозначающую символ "**U+004A** Latin Capital Letter **J**" в Юникоде, т.е. латинскую прописную букву **J**. Кодовую единицу можно выразить и в шестнадцатеричной форме с префиксом **\u**. Например, кодовая единица `'\u004A'` обозначает то же самое, что и символьный литерал `'J'`. В качестве более экзотичного примера можно привести кодовую единицу `'\u263A'`, обозначающую знак ☺, "**U+263A** White Smiling Face" в Юникоде, т.е. светлую улыбку.

Специальные символы `'\n'`, `'\r'`, `'\t'`, `'\b'` обозначают перевод строки, возврат каретки, табуляцию и "забой" (возврат на одну позицию назад) соответственно. Знак обратной косой черты служит для экранирования одиночной кавычки и самой обратной косой черты следующим образом: `'\''` и `'\\'`.

1.2.4. Логический тип

Логический тип `boolean` принимает два значения: `false` и `true`. В языке Java тип `boolean` не является числовым. Логические значения `false` и `true` типа `boolean` никак не связаны с целыми числами **0** и **1** соответственно.

1.3. Переменные

В последующих разделах поясняется, как объявлять и инициализировать переменные и константы.

1.3.1. Объявление переменных

Язык Java является строго типизированным. Каждая переменная может содержать значения только конкретного типа. При объявлении переменной требуется указать тип, имя и необязательное начальное значение, как показано в следующем примере кода:

```
int total = 0;
```

В одном операторе можно объявить сразу несколько переменных одного и того же типа, как показано ниже. Но большинство программирующих на Java предпочитают объявлять переменные по отдельности.

```
int total = 0, count; // переменная count инициализируется целым значением
```

Когда переменная объявляется и инициализируется построенным объектом, имя класса этого объекта указывается дважды, как показано в следующем примере кода:

```
Random generator = new Random();
```

Сначала имя `Random` обозначает в данном примере тип переменной `generator`, а затем употребляется как часть выражения с оператором `new` для построения объекта данного класса.

1.3.2. Именование переменных

Имя переменной (а также метода или класса) должно начинаться с буквы. Оно может состоять из любых букв, цифр и знаков _ и $. Но знак $ предназначен для обозначения автоматически генерируемых имен, и поэтому пользоваться им в именах переменных не рекомендуется.

В именах переменных могут быть указаны буквы и цифры из *любого* алфавита, а не только латинского. Например, имена переменных π и **élévation** являются вполне допустимыми. Но регистр символов в Java имеет значение. Так, имена **count** и **Count** обозначают разные переменные.

В именах переменных не разрешается употреблять пробелы или знаки, кроме упомянутых выше. И наконец, в качестве имени переменной нельзя употреблять ключевое слово, например `double`.

Имена переменных и методов принято начинать со строчной буквы, а имена классов — с прописной. Программирующие на Java предпочитают употреблять смешанное написание (так называемый "горбатый регистр"), где с прописных букв начинаются слова, составляющие имена, как, например, `countOfInvalidInputs`.

1.3.3. Инициализация переменных

Объявляя переменную в методе, следует инициализировать ее, прежде чем воспользоваться ею. Например, компиляция следующей строки кода приведет к ошибке:

```
int count;
count++; // ОШИБКА! Используется неинициализированная переменная
```

Компилятор должен быть в состоянии проверить, что переменная инициализирована до ее применения. Например, следующий фрагмент кода также содержит ошибку:

```
int count;
if (total == 0) {
  count = 0;
} else {
  count++; // ОШИБКА! Переменная count может быть не инициализирована
}
```

Переменную можно объявить в любом месте тела метода. Тем не менее хорошим стилем программирования считается объявлять переменную как можно позже, непосредственно перед ее употреблением в первый раз. Так, в следующем примере кода переменная объявляется в тот момент, когда оказывается доступным ее начальное значение:

```
System.out.println("How old are you?");
int age = in.nextInt(); // см. далее раздел 1.6.1
```

1.3.4. Константы

Ключевым словом `final` обозначается значение, которое не может быть изменено после его присваивания. В других языках программирования такое значение называется *константой*. Ниже приведен пример объявления целочисленной константы. Имена констант принято обозначать прописными буквами.

```
final int DAYS_PER_WEEK = 7;
```

Константу можно объявить и вне тела метода, используя ключевое слово `static` следующим образом:

```
public class Calendar {
  public static final int DAYS_PER_WEEK = 7;
  ...
}
```

Такой константой можно пользоваться в нескольких методах. В классе `Calendar` обращаться к этой константе можно по ее имени `DAYS_PER_WEEK`. А для того чтобы воспользоваться ею в другом классе, имя этой константы следует предварить именем ее класса следующим образом: `Calendar.DAYS_PER_WEEK`.

НА ЗАМЕТКУ. В классе **`System`** объявляется следующая константа:

```
public static final PrintStream out
```

Ею можно пользоваться везде как **`System.out`**. Это один из немногих примеров константы, имя которой не обозначается прописными буквами.

Откладывать инициализацию конечной переменной вполне допустимо, при условии, что она инициализируется непосредственно перед ее употреблением в первый раз. Например, следующий фрагмент кода считается допустимым:

```
final int DAYS_IN_FEBRUARY;
if (leapYear) {
  DAYS_IN_FEBRUARY = 29;
} else {
  DAYS_IN_FEBRUARY = 28;
}
```

Именно по этой причине такие переменные называются *конечными*. Как только такой переменной будет присвоено значение, оно станет конечным и не допускающим изменение в дальнейшем.

НА ЗАМЕТКУ. Иногда требуется задать ряд связанных вместе констант, как в следующем примере кода:

```
public static final int MONDAY = 0;
public static final int TUESDAY = 1;
...
```

В таком случае можно определить *перечислимый тип*, как показано ниже.

```
enum Weekday { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Перечисление **Weekday** состоит из значений перечислимого типа **Weekday.MON**, **Weekday.TUE** и т.д. А переменная перечислимого типа **Weekday** объявляется следующим образом:

```
Weekday startDay = Weekday.MON;
```

Более подробно перечислимые типы рассматриваются в главе 4.

1.4. Арифметические операции

В языке Java употребляются такие же операции, как и в C-подобных языках (табл. 1.3). Более подробно они рассматриваются в последующих разделах.

НА ЗАМЕТКУ. В табл. 1.3 операции перечислены по порядку снижения их *предшествования*. Так, операция + имеет более высокое предшествование, чем операция <<, и поэтому выражение `3 + 4 << 5` имеет значение `(3 + 4) << 5`. Операция считается с *левой ассоциативностью*, если она сгруппирована слева направо. Например, выражение `3 - 4 - 5` означает `(3 - 4) - 5`. Но операция `-=` оказывается с *правой ассоциативностью*, и поэтому выражение `i -= j -= k` означает `i -= (j -= k)`.

Таблица 1.3. Операции в Java

Операции	Ассоциативность
`[] . ()` (вызов метода)	Левая
`! ~ ++ --` + (унарный плюс) - (унарный минус) `()` (приведение типов) `new`	Правая
`* / %` (деление по модулю)	Левая
`+ -`	Левая
`<< >> >>>` (арифметический сдвиг)	Левая
`< > <= >= instanceof`	Левая
`== !=`	Левая
`&` (поразрядное И)	Левая
^ (поразрядное исключающее ИЛИ)	Левая
| (поразрядное ИЛИ)	Левая
`&&` (логическая операция И)	Левая
|| (логическая операция ИЛИ)	Левая
? : (условная операция)	Левая
`= += -= *= /= %= <<= >>= >>>= &= ^= |=`	Правая

1.4.1. Присваивание

В последней строке табл. 1.3 перечислены операции присваивания. Так, в следующем операторе:

```
x = выражение;
```

переменной x присваивается значение из правой части выражения, которое заменяет собой предыдущее. В сочетании с другой операцией такая операция объединяет левую и правую стороны выражения, присваивая результат. Например, операция присваивания

```
amount -= 10;
```

аналогична следующей:

```
amount = amount - 10;
```

1.4.2. Основные арифметические операции

Арифметические операции сложения, вычитания, умножения и деления обозначаются знаками **+ - * /** соответственно. Например, выражение **2 * n + 1** означает умножение **2** на **n** и прибавление **1**.

Выполняя операцию деления **/**, следует проявлять внимательность. Если оба ее операнда относятся к целочисленным типам данных, то такая операция обозначает целочисленное деление с отбрасыванием остатка. Например, результат деления **17 / 5** равен **3**, тогда как результат деления **17.0 / 5** равен **3.4**.

Целочисленное деление на нуль приводит к исключению, которое, в свою очередь, приводит к преждевременному завершению программы, если не перехватить его вовремя. (Подробнее об обработке исключений — в главе 5.) А деление на нуль чисел с плавающей точкой дает бесконечное значение или не число (NaN; см. раздел 1.2.2), не приводя к исключению.

Арифметическая операция **%** дает остаток от деления. Например, результат операции **17 % 5** равен **2**, т.е. величине остатка после вычитания из числа **17** числа **15** (наибольшего целочисленного множителя числа **5**, который "вмещается" в число **17**). Если же остаток от деления a % b равен нулю, то a оказывается целочисленным множителем b.

Арифметическая операция **%** чаще всего служит для проверки четности целочисленного значения. Так, результат вычисления выражения n % 2 равен **0**, если n имеет четное значение. Когда же значение n нечетное, результат вычисления выражения n % 2 равен **1**, если n имеет положительное значение, или **-1**, если n имеет отрицательное значение. Но такая обработка отрицательных чисел считается неудачным практическим приемом. Выполняя арифметическую операцию **%** с потенциально отрицательными операндами, следует всегда проявлять особую внимательность.

Рассмотрим следующее затруднение. Допустим, вычисляется положение часовой стрелки. Внося коррективы в положение часовой стрелки, требуется нормализовать число в пределах от **0** до **11**. Это нетрудно сделать в следующем выражении: (position

+ adjustment) % 12. Но что, если в результате внесения корректив (adjustment) положение часовой стрелки (position) становится отрицательным? В таком случае может получиться отрицательное число. Следовательно, в расчет положения часовой стрелки придется ввести переход по условию или воспользоваться выражением ((position + adjustment) % 12 + 12) % 12. Но и то и другое хлопотно.

СОВЕТ. В данном случае проще воспользоваться методом **Math.floorMod()**. Так, вызов **Math.floorMod(position + adjustment, 12)** всегда дает значение в пределах от 0 до 11. К сожалению, метод **floorMod()** дает отрицательные результаты для отрицательных делителей, хотя такое не часто встречается на практике.

В языке Java предусмотрены операции инкремента и декремента, как показано ниже.

```
n++; // Добавляет единицу к переменной n
n--; // Вычитает единицу из переменной n
```

Как и в других С-подобных языках программирования, у этих операций имеется также префиксная форма. В частности, обе операции, n++ и ++n, выполняют приращение переменной n на единицу, но они имеют разные значения, когда применяются в выражении. В первой форме операция инкремента дает значение до приращения на единицу, а во второй форме — значение после приращения на единицу. Например, в следующем выражении:

```
String arg = args[n++];
```

сначала переменной arg присваивается значение элемента массива args[n], а *затем* происходит приращение индекса массива n на единицу. В настоящее время разделение совмещенной операции инкремента на две отдельные операции не влечет за собой никакого снижения производительности, и поэтому многие программисты предпочитают употреблять более удобочитаемую форму данной операции.

НА ЗАМЕТКУ. Одной из заявленных целей разработки языка программирования Java была переносимость его кода. Вычисления должны давать одинаковые операции независимо от того, на какой виртуальной машине они выполняются. Но многие современные процессоры оснащены регистрами с плавающей точкой и разрядностью больше 64 бит, обеспечивая дополнительную точность и исключая риск переполнения на промежуточных стадиях вычисления. Подобные средства оптимизации вычислений допускаются в Java, ведь иначе операции с плавающей точкой выполнялись бы медленнее и менее точно. Для тех немногочисленных пользователей, кому это очень важно, в Java предусмотрен модификатор доступа **strictfp**. Если метод объявляется с этим модификатором доступа, то все операции с плавающей точкой в таком методе оказываются строго переносимыми.

1.4.3. Математические методы

Для возведения чисел в степень в Java отсутствует соответствующая операция. Вместо нее вызывается метод Math.pow(). Так, в результате вызова Math.pow(x, y) получается величина x^y. Для извлечения квадратного корня из величины x вызывается метод Math.sqrt(x). Эти методы являются *статическими* (static), т.е. они не

оперируют объектами. Подобно статическим константам, имена этих методов предваряются именем того класса, в котором они объявляются.

Для вычисления минимального и максимального значений полезными также оказываются соответствующие методы `Math.min()` и `Math.max()`. Кроме того, в классе `Math` предоставляются методы, реализующие тригонометрические и логарифмические функции, а также константы `Math.PI` и `Math.E` для обозначения числа пи и экспоненты соответственно.

НА ЗАМЕТКУ. В классе **`Math`** предоставляется несколько методов для более надежного выполнения арифметических операций. Математические операции негласно возвращают неверные результаты, если вычисление приводит к переполнению. Так, если умножить один миллиард на три (**`1000000000 * 3`**), в итоге получится значение **`-1294967296`**, поскольку самое большое значение типа **`int`** не превышает двух миллиардов. Если же вместо этого вызвать метод **`Math.multiplyExact(1000000000, 3)`**, то будет сгенерировано исключение. Это исключение можно перехватить или же допустить преждевременное завершение программы вместо того, чтобы спокойно продолжить ее выполнение с неверным результатом операции. В классе **`Math`** имеются также методы **`addExact()`**, **`subtractExact()`**, **`incrementExact()`**, **`decrementExact()`**, **`negateExact()`**, принимающие параметры типа **`int`** и **`long`**.

Некоторые математические методы имеются и в других классах. Например, в классах `Integer` и `Long` имеются методы `compareUnsigned()`, `divideUnsigned()` и `remainderUnsigned()` для обращения с целочисленными значениями без знака.

1.4.4. Преобразования числовых типов

Когда в операции сочетаются операнды разных числовых типов, числа преобразуются в общий тип перед их объединением. Такое преобразование происходит в следующем порядке.

1. Если один из операндов относится к типу `double`, то другой операнд преобразуется в тип `double`.
2. Если один из операндов относится к типу `float`, то другой операнд преобразуется в тип `float`.
3. Если один из операндов относится к типу `long`, то другой операнд преобразуется в тип `long`.
4. Иначе оба операнда преобразуются в тип `int`.

Так, если вычисляется выражение **`3.14 + 42`**, то второй его операнд преобразуется в значение **`42.0`**, а затем вычисляется полученная сумма, давая в итоге значение **`45.14`**. А если вычисляется выражение **`'J' + 1`**, то символьное значение **`'J'`** типа `char` преобразуется в числовое значение **`74`** типа `int`, а в итоге получается числовое значение **`75`** типа `int`. Далее в этой главе поясняется, как преобразовать это значение обратно в тип `char`.

Когда значение числового типа присваивается переменной или передается методу в качестве аргумента, а их типы не совпадают, то такое значение должно быть преобразовано в соответствующий тип. Например, в следующей операции присваивания:

```
double x = 42;
```

числовое значение **42** преобразуется из типа `int` в тип `double`.

В языке Java преобразование всегда допустимо, если оно не приводит к потере информации. В частности, вполне допустимы следующие преобразования.

- Из типа `byte` в тип `short`, `int`, `long` или `double`.
- Из типа `short` и `char` в тип `int`, `long` или `double`.
- Из типа `int` в тип `long` или `double`.

Преобразование из целочисленного типа в тип чисел с плавающей точкой всегда допустимо.

ВНИМАНИЕ. Следующие преобразования вполне допустимы, но могут привести к потере информации:

- Из типа **`int`** в тип **`float`**.
- Из типа **`long`** в тип **`float`** или **`double`**.

Рассмотрим в качестве примера следующую операцию присваивания:

```
float f = 123456789;
```

Тип **`float`** обеспечивает точность только до семи значащих цифр, и поэтому переменной **`f`** фактически присваивается числовое значение **`1.23456792E8`**.

Чтобы выполнить преобразование, не относящееся к категории перечисленных выше разрешенных преобразований, достаточно воспользоваться операцией приведения типов, заключив целевой тип данных в круглые скобки. Так, в следующем примере кода дробная часть числа отбрасывается и переменной `n` присваивается значение **3**:

```
double x = 3.75;
int n = (int) x;
```

Если же результат операции требуется округлить до ближайшего целого числа, то следует вызвать метод `Math.round()`, возвращающий значение типа `long`. Если заранее известно, что результат округления "вмещается" в тип `int`, то данный метод вызывается так, как показано ниже, где `x` = **`3.75`**, а переменной `n` присваивается значение **4**.

```
int n = (int) Math.round(x);
```

Чтобы преобразовать один целочисленный тип в другой с меньшим количеством байт, придется также воспользоваться приведением типов, как показано ниже.

```
char next = (char)('J' + 1); // Преобразует значение 75 в символ 'K'
```

При таком приведении типов сохраняются только последние байты.

```
int n = (int) 3000000000L; // Присваивает значение -1294967296 переменной n
```

НА ЗАМЕТКУ. Если существует опасность, что в результате приведения типов негласно будут отброшены важные части числа, в таком случае лучше воспользоваться методом **`Math.toIntExact()`**. Если этот метод не сумеет преобразовать тип **`long`** в тип **`int`**, возникнет исключение.

1.4.5. Операции отношения и логические операции

Операции **==** и **!=** служат для проверки на равенство и неравенство соответственно. Например, в результате выполнения операции `n != 0` получается логическое значение `true`, если `n` не равно нулю.

В языке Java предоставляются также обычные операции отношения **<** (меньше), **>** (больше), **<=** (меньше или равно) и **>=** (больше или равно). Выражения типа `boolean` можно сочетать с логическими операциями **&&** (И), **||** (ИЛИ) и **!** (НЕ). Так, если в приведенном ниже выражении значение `n` находится в пределах от нуля (включительно) до значения `length` (исключительно), то в результате вычисления этого выражения получается логическое значение `true`.

```
0 <= n && n < length
```

Если первое условие ложно (`false`), то второе не вычисляется. Такое "укороченное" вычисление оказывается удобным, когда второе условие может привести к ошибке. Рассмотрим следующее выражение:

```
n != 0 && s + (100 - s) / n < 50
```

Если значение `n` равно нулю, то второе условие, содержащее операцию деления на `n`, вообще не вычисляется. Следовательно, ошибка не возникает.

Укороченное вычисление применяется и в логических операциях ИЛИ, но в этом случае вычисление прекращается, как только первый операнд принимает логическое значение `true`. Например, вычисление следующего выражения:

```
n == 0 || s + (100 - s) / n >= 50
```

дает логическое значение `true`, если `n` равно нулю. И в этом случае второе условие не вычисляется.

И наконец, *условная* операция принимает следующие три операнда: условие и два значения. Если условие принимает логическое значение `true`, то в итоге выбирается первое значение, а иначе — второе. Так, в следующем примере кода:

```
time < 12 ? "am" : "pm"
```

получается символьная строка `"am"`, если `time < 12`, а иначе — символьная строка `"pm"`.

НА ЗАМЕТКУ. В языке Java предоставляются также поразрядные логические операции **&** (И), **|** (ИЛИ) и **^** (исключающее ИЛИ). Они выполняются над битовыми комбинациями целочисленных значений. Например, в результате выполнения операции **`n & 0xF`** получаются четыре младших бита значения **`n`**, поскольку шестнадцатеричное значение **`0xF`** содержит двоичные разряды `0...01111`; в результате выполнения операции **`n = n | 0xF`** в четырех младших битах устанавливается `1`; а в

результате выполнения операции `n = n ^ 0xF` их состояние меняется на обратное. Поразрядная операция ~ служит аналогом логической операции `!`, меняя на обратное состояние всех битов ее операнда. Так, результат выполнения операции `~0xF` равен `1...10000`.

Имеются также операции сдвига битовой комбинации влево или вправо. Например, в результате выполнения операции `0xF << 2` получаются двоичные разряды `0...0111100`. Для сдвига вправо имеются две операции. Так, операция `>>` заполняет старшие биты нулями, а операция `>>>` сдвигает знаковый бит в старшие биты. Эти операции имеют значение для тех, кому требуется манипулировать отдельными битами в своих программах, а остальным они вообще не нужны.

ВНИМАНИЕ. Правый операнд в операциях сдвига сокращается по модулю 32, если левый оператор относится к типу `int`, или же по модулю 64, если левый оператор относится к типу `long`. Например, выполнение операции `1 << 35` дает такое же значение `8`, как и выполнение операции `1 << 3`.

СОВЕТ. Поразрядные логические операции `&` (И) и `|` (ИЛИ) над значениями типа `boolean` обусловливают принудительное вычисление обоих операндов перед объединением результатов. Такое применение этих операций не совсем обычно. Если у правого операнда отсутствует побочный эффект, эти операции действуют подобно логическим операциям `&&` и `||`, за исключением того, что они менее эффективны. Если же действительно требуется принудительное вычисление второго операнда, его следует присвоить переменной типа `boolean`, чтобы стал более ясным порядок выполнения операции.

1.4.6. Большие числа

Если точности примитивных целочисленных и числовых типов с плавающей точкой недостаточно, можно прибегнуть к помощи классов `BigInteger` и `BigDecimal` из пакета `java.math`. Объекты этих классов представляют числа с произвольно длинной последовательностью цифр. Так, в классе `BigInteger` реализуется целочисленная арифметика произвольной точности, а в классе `BigDecimal` делается то же самое в отношении чисел с плавающей точкой.

Статический метод `valueOf()` возвращает значение типа `long` объекту типа `BigInteger` следующим образом:

```
BigInteger n = BigInteger.valueOf(876543210123456789L);
```

Объект типа `BigInteger` можно также построить из символьной строки цифр:

```
BigInteger k = new BigInteger("9876543210123456789");
```

В языке Java не разрешается выполнять операции над объектами, и поэтому для обращения с большими числами приходится вызывать соответствующие методы:

```
BigInteger r = BigInteger.valueOf(5).multiply(n.add(k));
// что равнозначно операции r = 5 * (n + k)
```

Как было показано в разделе 1.2.2, результат вычитания чисел с плавающей точкой `2.0 - 1.1` равен `0.8999999999999999`. Класс `BigDecimal` позволяет вычислить результат подобной операции более точно.

В результате вызова метода `BigDecimal.valueOf(n, e)` возвращается экземпляр класса `BigDecimal` со значением $n \times 10^{-e}$. А в результате вызова

```
BigDecimal.valueOf(2, 0).subtract(BigDecimal.valueOf(11, 1))
```

возвращается значение ровно `0.9`.

1.5. Символьные строки

Символьная строка представляет собой последовательность символов. В языке Java строки могут состоять из любых символов в Юникоде. Например, символьная строка "Java™" или "Java\u2122" состоит из следующих пяти символов: J, a, v, a и ™. Последний символ называется "U+2122 Trade Mark Sign", т.е. знак торговой марки.

1.5.1. Сцепление символьных строк

Для сцепления двух символьных строк служит операция +. Так, в следующем примере кода:

```
String location = "Java";
String greeting = "Hello " + location;
```

переменной `greeting` присваивается сцепленная символьная строка `"Hello Java"`. (Обратите внимание на пробел в конце первого операнда операции сцепления символьных строк.)

Если символьная строка сцепляется со значением другого типа, последнее преобразуется в символьную строку. Так, в результате выполнения приведенного ниже примера кода получается символьная строка `"42 years"`.

```
int age = 42;
String output = age + " years";
```

ВНИМАНИЕ. Сочетание операции сцепления и сложения может привести к неожиданным результатам. Так, в следующем примере кода:

```
"Next year, you will be " + age + 1 // ОШИБКА!
```

сначала выполняется сцепление символьной строки со значением переменной **age**, а затем прибавление **1**. В итоге получается символьная строка **"Next year, you will be 421"**. В подобных случаях следует употреблять круглые скобки, как показано ниже.

```
"Next year, you will be " + (age + 1) // Верно!
```

Чтобы объединить несколько символьных строк с разделителем, достаточно вызвать метод `join()` следующим образом:

```
String names = String.join(", ", "Peter", "Paul", "Mary");
    // Задает в строке имена "Peter, Paul, Mary"
```

В качестве первого аргумента данного метода указывается символьная строка разделителя, а далее следуют объединяемые символьные строки. Их можно указать

сколько угодно или же предоставить массив символьных строк. (Подробнее о массивах речь пойдет далее, в разделе 1.8.)

Иногда сцеплять большое количество символьных строк неудобно и неэффективно, если нужно лишь получить конечный результат. В таком случае лучше воспользоваться классом `StringBuilder`, как показано ниже.

```
StringBuilder builder = new StringBuilder();
while (дополнительные строки) {
  builder.append(следующая строка);
}
String result = builder.toString();
```

1.5.2. Подстроки

Чтобы разделить символьные строки, достаточно вызвать метод `substring()`, как показано в следующем примере кода:

```
String greeting = "Hello, World!";
String location = greeting.substring(7, 12); // Задает местоположение
                                             // символьной строки "World"
```

В качестве первого аргумента метода `substring()` указывается начальная позиция извлекаемой подстроки. Позиции подстрок начинаются с нуля.

А в качестве второго аргумента данного метода указывается первая позиция, которая не должна быть включена в извлекаемую подстроку. В рассматриваемом здесь примере на позиции **12** находится знак **!**, который не требуется включать в извлекаемую подстроку. Было бы любопытно указать ненужную позицию в строке, но при этом теряется следующее преимущество: разность **12 - 7** дает длину подстроки.

Иногда требуется извлечь все подстроки из символьной строки с разграничителями ее составляющих. Эту задачу позволяет решить метод `split()`, возвращающий массив подстрок:

```
String names = "Peter, Paul, Mary";
String[] result = names.split(", ");
    // Массив из трех строк ["Peter", "Paul", "Mary"]
```

В качестве разделителя символьной строки может быть указано любое регулярное выражение (подробнее об этом — в главе 9). Например, в результате вызова `input.split("\\s+")` введенная строка `input` разделяется по пробелам.

1.5.3. Сравнение символьных строк

Для проверки двух символьных строк на равенство служит метод `equals()`. Так, в следующем примере кода:

```
location.equals("World")
```

получается логическое значение `true`, если объектная переменная `location` действительно содержит символьную строку `"World"`.

ВНИМАНИЕ. Для сравнения символьных строк ни в коем случае нельзя пользоваться операцией ==. Так, в результате следующего сравнения:

```
location == "World" // Нельзя!
```

возвращается логическое значение **true**, если объектная переменная **location** и символьная строка **"World"** относятся к *одному и тому же объекту* в оперативной памяти. В виртуальной машине имеется лишь один экземпляр каждой литеральной строки, и поэтому сравнение **"World" == "World"** даст в итоге логическое значение **true**. Но если объектная переменная **location** была вычислена, например, следующим образом:

```
String location = greeting.substring(7, 12);
```

то полученный результат размещается в отдельном объекте типа **String**, а в результате сравнения **location == "World"** возвращается логическое значение **false**!

Переменная типа `String`, как и любая другая объектная переменная, может содержать пустое значение `null`, как показано ниже. Это означает, что переменная вообще не ссылается ни на один из объектов — даже на пустую символьную строку.

```
String middleName = null;
```

Чтобы проверить, является ли объект пустым (`null`), можно воспользоваться операцией == следующим образом:

```
if (middleName == null) ...
```

Следует иметь в виду, что пустое значение `null` не равнозначно пустой символьной строке `""`. Пустая символьная строка имеет нулевую длину, тогда как пустое значение `null` вообще не имеет никакого отношения к символьным строкам.

ВНИМАНИЕ. Вызов любого метода по пустой ссылке **null** приводит к "исключению в связи с пустым указателем". Как и все остальные исключения, оно приводит к преждевременному завершению программы, если не обработать его явным образом.

СОВЕТ. При сравнении символьной строки с литеральной строкой рекомендуется указывать литеральную строку первой, как показано ниже.

```
if ("World".equals(location)) ...
```

Эта проверка действует верно, даже если объектная переменная **location** содержит пустое значение **null**.

Для сравнения двух символьных строк без учета регистра символов служит метод `equalsIgnoreCase()`. Так, в следующем примере кода:

```
location.equalsIgnoreCase("world")
```

возвращается логическое значение `true`, если объектная переменная `location` содержит символьную строку `"World"`, `"world"` или `"WORLD"` и т.д. Иногда символьные строки требуется расположить в определенном порядке. Вызвав метод `compareTo()`,

можно определить лексикографический порядок следования символьных строк. Так, в результате следующего вызова:

```
first.compareTo(second)
```

возвращается отрицательное целочисленное значение (не обязательно **-1**), если строка `first` следует перед строкой `second`; положительное целочисленное значение (не обязательно **1**), если строка `first` следует после строки `second`; и нулевое значение, если обе сравниваемые строки равны.

Строки сравниваются посимвольно до тех пор, пока в одной из них не исчерпаются символы или будет обнаружено несовпадение. Например, при сравнении строк `"word"` и `"world"` первые три символа совпадают. Но поскольку значение символа `d` в Юникоде меньше, чем у символа `l`, то строка `"word"` следует первой. (В данном случае метод `compareTo()` возвращает значение **-8**, т.е. разность значений символов `d` и `l` в Юникоде.)

Такое сравнение может показаться нелогичным, поскольку оно зависит от значений символов в Юникоде. Так, символьная строка `"blue/green"` предшествует строке `"bluegreen"`, поскольку значение знака **/** в Юникоде меньше, чем у символа `g`.

СОВЕТ. Для сортировки удобочитаемых символьных строк служит объект типа **`Collator`**, которому известны правила сортировки, принятые в данном языке программирования. Подробнее об этом речь пойдет в главе 13.

1.5.4. Взаимное преобразование чисел и символьных строк

Чтобы преобразовать целое число в символьную строку, достаточно вызвать статический метод `Integer.toString()` следующим образом:

```
int n = 42;
String str = Integer.toString(n); // Задает строку "42" в переменной str
```

Имеется вариант этого метода, принимающий в качестве второго параметра основание системы счисления в пределах от **2** до **36**:

```
str = Integer.toString(n, 2); // Задает строку "101010" в переменной str
```

НА ЗАМЕТКУ. Еще более простой способ преобразовать число в символьную строку состоит в том, чтобы сцепить его с пустой строкой следующим образом: **`"" + n`**. Некоторым такой способ кажется скверным и менее эффективным.

С другой стороны, чтобы преобразовать в число символьную строку, содержащую целочисленное значение, можно вызвать метод `Integer.parseInt()` следующим образом:

```
n = Integer.parseInt(str); // Задает число 101010 в переменной n
```

При этом можно также указать основание системы счисления таким образом:

```
n = Integer.parseInt(str, 2); // Задает число 42 в переменной n
```

Для преобразования символьных строк в числа с плавающей точкой служат методы `Double.toString()` и `Double.parseDouble()`.

```
String str = Double.toString(3.14); // Задает строку "3.14" в переменной str
double x = Double.parseDouble("3.14"); // Задает число3.14 в переменной x
```

1.5.5. Прикладной программный интерфейс API для обработки символьных строк

Как и следовало ожидать, в классе `String` имеется немало методов. Некоторые из наиболее употребительных методов для обработки символьных строк приведены в табл. 1.4.

Таблица 1.4. Наиболее употребительные методы для обработки символьных строк в Java

Метод	Назначение
`boolean startsWith(String str)`	Проверяют, начинается ли,
`boolean endsWith(String str)`	оканчивается ли и содержит ли символьная строка
`boolean contains(CharSequence str)`	заданную строку ***`str`***
`int indexOf(String str)`	Получают позицию первого или последнего вхождения заданной строки ***`str`***,
`int lastIndexOf(String str)`	выполняя поиск всей строки или подстроки, начиная с указанного индекса ***`fromIndex`***
`int indexOf(String str, int fromIndex)`	Возвращают значение **`-1`**,
`int lastIndexOf(String str, int fromIndex)`	если совпадение не обнаружено
`String replace(CharSequence oldString, CharSequence newString)`	Возвращает символьную строку, получаемую заменой всех вхождений прежней строки ***`oldString`*** в новой строке ***`newString`***
`String toUpperCase()`	Возвращают строку, состоящую из всех символов исходной строки,
`String toLowerCase()`	преобразованных в верхний или нижний регистр символов
`String trim()`	Возвращает символьную строку, получаемую удалением всех начальных и конечных пробелов

Следует иметь в виду, что в Java класс `String` является *неизменяемым*. Это означает, что ни один из методов этого класса не видоизменяет символьную строку, которой он оперирует. Так, в следующем примере кода:

```
greeting.toUpperCase()
```

возвращается *новая* символьная строка `"HELLO, WORLD!"`, не изменяя исходную строку `greeting`.

Следует также иметь в виду, что у некоторых методов из класса `String` имеются параметры типа `CharSequence`. Это общий подтип классов `String`, `StringBuilder` и прочих последовательностей символов.

Более подробное описание каждого метода можно найти в документации на прикладной программный интерфейс Java API, доступной по адресу `http://docs.oracle.com/javase/8/docs/api`. На рис. 1.3 показано, как перемещаться по этой документации.

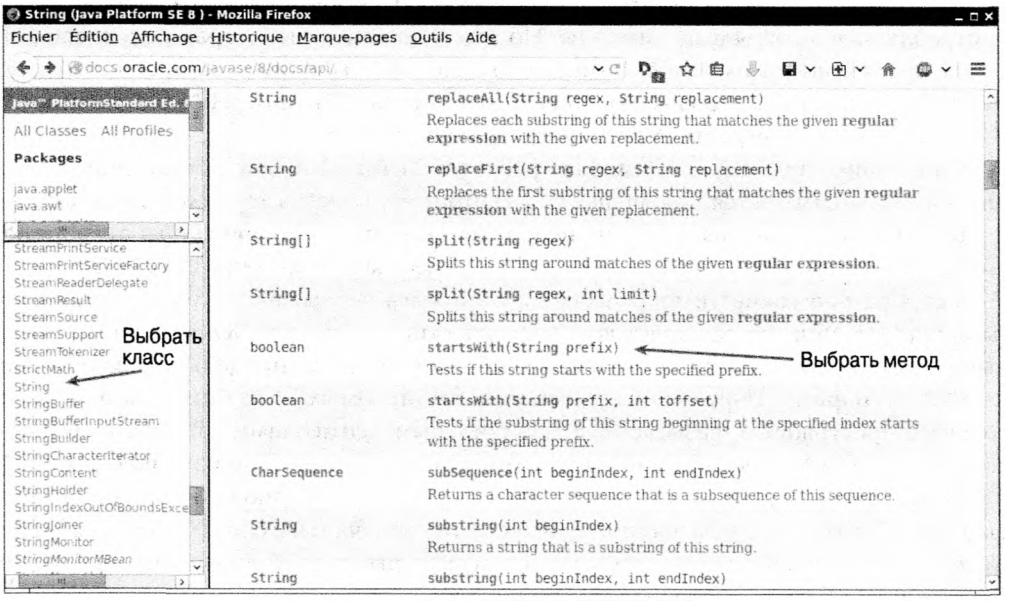


Рис. 1.3. Перемещение по документации на прикладной программный интерфейс Java API

В этой книге прикладной программный интерфейс Java API не представлен во всех подробностях, поскольку его проще просматривать в упомянутой выше документации. Если же у вас нет постоянного подключения к Интернету, можете загрузить и разархивировать эту документацию для просмотра в автономном режиме.

1.5.6. Кодовые точки и кодовые единицы

С самого начала в Java соблюдался стандарт на Юникод, разработанный немного раньше. Этот стандарт был разработан с целью разрешить неприятный вопрос кодировки символов. До появления Юникода существовало много несовместимых кодировок символов. Для английского языка имелось едва ли не повсеместное согласие по поводу стандартного 7-разрядного кода ASCII, присваивавшего коды в пределах от **0** до **127** всем буквам английского алфавита, десятичным цифрам и многим знакам. В Западной Европе код ASCII был расширен до 8-разрядного кода, содержавшего символы с ударениями вроде **ä** и **é**. Но в России код ASCII был расширен до букв кириллицы на позициях от **128** до **255**. А в Японии применялась кодировка переменной длины для кодирования букв английского и японского алфавитов. То же самое

было сделано и в остальных странах. В итоге обмен текстовыми файлами в разных кодировках превратился в серьезную проблему.

Юникод призван устранить эту проблему присваиванием каждому символу во всех изобретенных до сих пор системах письма однозначного 16-разрядного кода в пределах от **0** до **65535**. В 1991 году была выпущена версия 1.0 Юникода, в которой использовалось чуть меньше половины из **65535** доступных кодовых значений. С самого начала в языке Java применялись 16-разрядные символы Юникода, что было существенным прогрессом по сравнению с другими языками программирования, где употреблялись 8-разрядные символы. Но затем возникла некоторая неловкость. На самом деле символов оказалось намного больше, чем было оценено предварительно, и это были, главным образом, китайские иероглифы. Это потребовало расширить Юникод далеко за пределы 16-разрядного кода символов.

В настоящее время для Юникода требуется 21 бит. Каждое достоверное значение Юникода называется *кодовой точкой*. Например, кодовая точка латинской буквы **A** равна **U+0041**, а математический знак © для обозначения множества октонионов (`http://math.ucr.edu/home/baez/octonions`) имеет кодовую точку **U+1D546**.

Для обратной совместимости имеется кодировка переменной длины, которая называется UTF-16 и представляет все "классические" символы в Юникоде единственным 16-разрядным значением, а символы, выходящие за пределы кодовой точки **U+FFFF**, — парами 16-разрядных значений, выбираемых из специальной области кодового пространства, называемого "суррогатными символами". В этой кодировке латинская буква **A** имеет код **\u0041**, а математический знак © — код **\ud835\udd46**.

Языку Java не повезло в том, что он появился на свет в период перехода от 16-разрядного Юникода к 21-разрядному. Вместо первоначальных последовательностей символов в Юникоде (или кодовых точек) символьные строки в Java состоят из *кодовых единиц* — 16-разрядных величин кодировки UTF-16.

Если вас не особенно волнуют китайские иероглифы и вы готовы отказаться от таких специальных знаков, как ©, то можете и дальше тешить себя иллюзией, что объект типа `String` в Java состоит из последовательности символов в Юникоде. В таком случае *i*-й символ в строке можно получить следующим образом:

```
char ch = str.charAt(i);
```

а длину символьной строки так:

```
int length = str.length();
```

Но если вам требуется обрабатывать символьные строки надлежащим образом, то для этого придется немного потрудиться. В частности, чтобы получить *i*-ю кодовую точку в Юникоде, нужно сделать следующий вызов:

```
int codePoint = str.codePointAt(str.offsetByCodePoints(0, i));
```

Общее количество кодовых точек получается следующим образом:

```
int length = str.codePointCount(0, str.length());
```

Если в прикладном коде выполняется обход символьной строки и требуется проанализировать каждую кодовую точку по очереди, то для этой цели следует вызвать

метод codePoints(), предоставляющий *поток данных* со значениями типа int для каждой кодовой точки. Более подробно потоки данных рассматриваются в главе 8, а до тех пор достаточно сказать, что полученный поток данных можно преобразовать в массив, как показано ниже, а затем выполнить его обход.

```
int[] codePoints = str.codePoints().toArray();
```

1.6. Ввод-вывод

Чтобы сделать программы интереснее, следует организовать их взаимодействие с пользователем. В последующих разделах будет показано, как читать вводимые с терминала данные и выводить на него отформатированные данные.

1.6.1. Чтение вводимых данных

Когда вызывается метод System.out.println(), выводимые данные посылаются в так называемый "стандартный поток вывода" и появляются в окне терминала. Но прочитать данные из так называемого "стандартного потока ввода" сложнее, поскольку у соответствующего объекта System.in имеются методы только для чтения отдельных байтов. Чтобы читать символьные строки и числа, придется построить объект типа Scanner, присоединяемый к объекту System.in, представляющему стандартный поток ввода, следующим образом:

```
Scanner in = new Scanner(System.in);
```

Метод nextLine() служит для чтения строки из вводимых символьных данных, как показано ниже.

```
System.out.println("What is your name?");
String name = in.nextLine();
```

В данном случае имеет смысл воспользоваться методом nextLine(), поскольку вводимые символьные данные могут содержать пробелы. Чтобы прочитать отдельное слово, отделяемое пробелом, достаточно сделать следующий вызов:

```
String firstName = in.next();
```

Чтобы прочитать целочисленное значение, достаточно вызвать метод nextInt(), как показано ниже. Аналогичным образом метод nextDouble() вызывается для чтения следующего числового значения с плавающей точкой.

```
System.out.println("How old are you?");
int age = in.nextInt();
```

С помощью методов hasNextLine(), hasNext(), hasNextInt() и hasNextDouble() можно проверить наличие для ввода очередной строки, слова, целочисленного или числового значения с плавающей точкой, как показано ниже.

```
if (in.hasNextInt()) {
  int age = in.nextInt();
```

```
   ...
}
```

Класс Scanner входит в состав пакета java.util. Чтобы воспользоваться этим классом, достаточно ввести в начале исходного файла программы следующую строку кода:

```
import java.util.Scanner
```

СОВЕТ. Пользоваться средствами класса **Scanner** для чтения пароля не рекомендуется, поскольку вводимые данные в этом случае видны на терминале. Вместо этого лучше воспользоваться классом **Console** следующим образом:

```
Console terminal = System.console();
String username = terminal.readLine("User name: ");
char[] passwd = terminal.readPassword("Password: ");
```

Пароль возвращается в виде массива символов. Это несколько более безопасный способ, чем сохранение пароля в объекте типа String, поскольку массив можно перезаписать по завершении его обработки.

СОВЕТ. Если требуется прочитать вводимые данные из файла или записать выводимые данные в файл, то для этой цели можно воспользоваться синтаксисом переадресации применяемого командного процессора, как показано ниже.

```
java mypackage.MainClass < input.txt > output.txt
```

Теперь данные читаются в стандартный поток ввода **System.in** из файла **input.txt** и записываются в файл **output.txt** из стандартного потока вывода **System.out**. В главе 9 будет показано, как выполнять более общие операции файлового ввода-вывода.

1.6.2. Форматированный вывод данных

Ранее было показано, как пользоваться методом println() объекта System.out для построчного вывода данных. Имеется также метод print(), который не начинает вывод с новой строки. Зачастую он служит для вывода приглашений на ввод информации, как показано ниже. В этом случае курсор останавливается после выведенного приглашения, а не переводится на новую строку.

```
System.out.print("Your age: "); // Это не метод println()!
int age = in.nextInt();
```

Если дробное число выводится методом print() или println(), то отображаются все его цифры, кроме конечных нулей. Так, в следующем примере кода:

```
System.out.print(1000.0 / 3.0);
```

выводится такой результат:

```
333.3333333333333
```

Но вывести денежную сумму в конкретной валюте, например, в долларах и центах, не так-то просто. Чтобы ограничить количество цифр в выводимом числе, можно воспользоваться методом `printf()` следующим образом:

```
System.out.printf("%8.2f", 1000.0 / 3.0);
```

Форматирующая строка `"%8.2f"` обозначает, что число с плавающей точкой выводится с *шириной поля* **8** и *точностью* до **2** цифр. Это означает, что выводимый результат содержит два начальных пробела и шесть символов:

```
333.33
```

Метод `printf()` принимает немало других параметров, как показано в следующем примере кода:

```
System.out.printf("Hello, %s. Next year, you'll be %d.\n", name, age);
```

Каждый *спецификатор формата*, начинающийся со знака **%**, заменяется соответствующим аргументом. *Символ преобразования*, завершающий спецификатор формата, обозначает тип форматируемого значения: **f** — число с плавающей точкой, **s** — символьную строку, **d** — десятичное целое число. Все символы преобразования приведены в табл. 1.5.

Таблица 1.5. Символы преобразования для форматированного вывода

Символ преобразования	Назначение	Пример
d	Десятичное целое число	**159**
x или **X**	Шестнадцатеричное целое число	**9f** или **9F**
o	Восьмеричное целое число	**237**
f или **F**	Число с плавающей точкой фиксированной точности	**15.9**
e или **E**	Число с плавающей точкой в экспоненциальной форме	**1.59e+01** или **1.59E+01**
g или **G**	Число с плавающей точкой в общей форме: короче, чем в форме **e/E** и **f/F**	—
a или **A**	Шестнадцатеричное число с плавающей точкой	**0x1.fccdp3** или **0X1.FCCDP3**
s или **S**	Символьная строка	**Java** или **JAVA**
c или **C**	Символ	**j** или **J**
b или **B**	Логическое значение	**false** или **FALSE**
h или **H**	Хеш-код (см. главу 4)	**42628b2** или **42628B2**
t или **T**	Дата и время (устаревшее; замену см. в главе 12)	—
%	Знак процента	**%**
n	Разделитель строк, зависящий от конкретной платформы	—

Кроме того, можно указывать признаки для управления внешним видом форматируемого вывода. Все имеющиеся признаки перечислены в табл. 1.6. Например, признак , (запятая) вводит разделители групп, а признак **+** (знак "плюс") снабжает

соответствующим знаком положительные и отрицательные числа. Так, следующий оператор:

```
System.out.printf("%,+.2f", 100000.0 / 3.0);
```

выводит такой результат:

```
+33,333.33
```

Для составления отформатированной символьной строки, не выводя ее, можно воспользоваться методом `String.format()` следующим образом:

```
String message = String.format("Hello, %s. Next year,
                               you'll be %d.\n", name, age);
```

Таблица 1.6. Признаки для форматированного вывода

Признак	Назначение	Пример
`+`	Снабжает соответствующим знаком положительные и отрицательные числа	`+3333.33`
Пробел	Вводит пробел перед положительными числами	`_3333.33`
`-`	Выравнивает поле по левому краю	`3333.33___`
`0`	Вводит начальные нули	`003333.33`
`(`	Заключает отрицательные числа в круглые скобки	`(3333.33)`
`,`	Вводит разделители групп	`3,333.33`
`#` (для формата `f`)	Всегда включает десятичную точку	`3333.`
`#` (для формата `x` или `o`)	Вводит префикс `0x` или `0`	`0xcafe`
`$`	Обозначает индекс форматируемого аргумента; например, спецификация `%1$d %1$x` означает вывод первого аргумента в десятичной и шестнадцатеричной форме	`159 9f`
`<`	Форматирует то же самое значение, как и в предыдущей спецификации; например, спецификация `%d %<x` означает вывод одного и того же числа в десятичной и шестнадцатеричной форме	`159 9f`

1.7. Управляющая логика

В последующих разделах будет показано, каким образом реализуются условные переходы и циклы. Синтаксис Java для операторов управляющей логики очень похож на аналогичный синтаксис, применяемый в других языках программирования, в том числе C/C++ и JavaScript.

1.7.1. Условные переходы

Условный оператор `if` снабжается условием в круглых скобках, после которого следует один оператор или целая группа операторов, заключаемых в фигурные скобки, как показано ниже.

```
if (count > 0) {
  double average = sum / count;
  System.out.println(average);
}
```

Этот оператор может быть дополнен условным переходом `else`, который происходит в том случае, если заданное условие не выполняется. Ниже приведен характерный тому пример.

```
if (count > 0) {
  double average = sum / count;
  System.out.println(average);
} else {
  System.out.println(0);
}
```

Оператор в условном переходе `else` может содержать другой условный оператор `if`, как показано ниже.

```
if (count > 0) {
  double average = sum / count;
  System.out.println(average);
} else if (count == 0) {
  System.out.println(0);
} else {
  System.out.println("Huh?");
}
```

Если требуется проверить выражение на соответствие конечному количеству константных значений, то можно воспользоваться оператором `switch`:

```
switch (count) {
  case 0:
    output = "None";
    break;
  case 1:
    output = "One";
    break;
  case 2:
  case 3:
  case 4:
  case 5:
    output = Integer.toString(count);
    break;
  default:
    output = "Many";
    break;
}
```

Выполнение начинается с совпадающей метки `case`, а в отсутствие совпадения — с метки `default`, если таковая имеется. Все операторы выполняются вплоть до оператора `break` или достижения конца оператора `switch`.

ВНИМАНИЕ. Нередко в конце альтернативной ветви оператора **`switch`** забывают указать оператор **`break`**. В таком случае выполнение "проваливается" к следующей альтернативной ветви.

Компилятор Java можно настроить на выявление подобных программных ошибок, указав соответствующий параметр в командной строке следующим образом:

```
javac -Xlint:fallthrough mypackage/MainClass.java
```

Если указан этот параметр, компилятор будет выдавать предупреждающее сообщение всякий раз, когда в альтернативной ветви оператора **switch** отсутствует оператор **break** или **return**. Если же режим "проваливания" действительно требуется, то метод, в теле которого присутствует оператор **switch**, следует снабдить *аннотацией* **@SuppressWarnings("fallthrough")**. И тогда компилятор не выдаст предупреждение для данного метода. (Аннотация снабжает информацией компилятор или другое инструментальное средство. Подробнее об аннотациях речь пойдет в главе 11.)

В предыдущем примере кода метки case были обозначены целыми числами. Для обозначения этих меток могут быть также использованы значения любого из следующих типов.

- Константное выражение типа char, byte, short или int (либо объекты соответствующих классов-оболочек Character, Byte, Short и Integer, которые будут представлены в разделе 1.8.3).
- Строковый литерал.
- Значение перечислимого типа (подробнее о нем — в главе 4).

1.7.2. Циклы

Тело цикла while продолжает выполняться до тех пор, пока требуются действия, определяемые условием данного цикла. Рассмотрим в качестве примера задачу суммирования чисел до тех пор, пока сумма не достигнет заданной величины, соответствующей намеченной цели. В качестве источника чисел воспользуемся генератором случайных чисел, предоставляемым классом Random из пакета java.util, как показано ниже.

```
Random generator = new Random();
```

В результате следующего вызова получается случайное число в пределах от 0 до 9:

```
int next = generator.nextInt(10);
```

Ниже приведен цикл для формирования суммы чисел.

```
while (sum < target) {
  int next = generator.nextInt(10);
  sum += next;
  count++;
}
```

Это типичный пример применения цикла while. До тех пор, пока значение переменной sum (сумма) меньше значения переменной target (цель), цикл while продолжает выполняться.

Иногда тело цикла требуется выполнить до того, как будет вычислено его условие. Допустим, требуется выяснить, сколько времени нужно для получения конкретного значения. Прежде чем проверить это условие, нужно войти в цикл и получить

значение. В таком случае следует воспользоваться циклом do/while, как показано ниже.

```
int next;
do {
  next = generator.nextInt(10);
  count++;
} while (next != target);
```

Сначала осуществляется вход в цикл и устанавливается значение переменной next. Затем вычисляется условие. Цикл повторяется до тех пор, пока это условие выполняется.

В предыдущих примерах количество шагов цикла заранее не было известно. Но во многих циклах, организуемых на практике, количество шагов цикла фиксировано. В подобных случаях лучше воспользоваться циклом for. В следующем примере цикла вычисляется сумма фиксированного количества случайных значений:

```
for (int i = 1; i <= 20; i++) {
  int next = generator.nextInt(10);
  sum += next;
}
```

Этот цикл выполняется 20 раз, причем на каждом его шаге в переменной i последовательно устанавливаются числовые значения 1, 2, . . . , 20. Любой цикл for можно переписать в виде цикла while. Так, приведенному выше циклу for равнозначен следующий цикл while:

```
int i = 1;
while (i <= 20) {
  int next = generator.nextInt(10);
  sum += next;
  i++;
}
```

Но в цикле while инициализация, проверка и обновление переменной цикла i рассредоточены в разных местах, тогда как в цикле for они находятся в одном конкретном месте. Инициализация, проверка и обновление переменной цикла может принимать произвольные формы. Например, значение переменной цикла можно удваивать до тех пор, пока оно меньше поставленной цели, как показано ниже.

```
for (int i = 1; i < target; i *= 2) {
  System.out.println(i);
}
```

Вместо объявления переменной цикла for в его заголовке можно инициализировать уже имеющуюся переменную следующим образом:

```
for (i = 1; i <= target; i++) // Используется уже имеющаяся переменная i
```

В цикле можно объявить, инициализировать и обновлять несколько переменных, разделив их запятыми, как показано в следующем примере кода:

```
for (int i = 0, j = n - 1; i < j; i++, j--)
```

Если инициализация или обновление не требуется, эти части цикла можно оставить пустыми. А если вообще опустить условие выполнения цикла, то оно будет всегда считаться истинным, как показано ниже. В следующем разделе будет показано, как прервать такой цикл и выйти из него.

```
for (;;) // Бесконечный цикл
```

1.7.3. Прерывание и продолжение цикла

Если требуется выйти из цикла посредине его выполнения, то для этой цели можно воспользоваться оператором `break`. Допустим, что слова требуется обрабатывать до тех пор, пока пользователь не введет букву `Q`. Ниже приведено решение этой задачи, где переменная типа `boolean` используется для управления циклом.

```
boolean done = false;
while (!done) {
  String input = in.next();
  if ("Q".equals(input)) {
    done = false;
  } else {
    Обработать переменную input
  }
}
```

В приведенном ниже примере цикла та же самая задача выполняется с помощью оператора `break`. По достижении оператора `break` происходит немедленный выход из цикла.

```
while (true) {
  String input = in.next();
  if (input.equals("Q")) break; // Выход из цикла
  Обработать переменную input
  }
// Сюда происходит переход по прерыванию цикла
```

Оператор `continue` действует аналогично оператору `break`, но вместо перехода в конец цикла он осуществляет переход в конец текущего шага цикла. С помощью этого оператора можно пропустить ненужные вводимые данные, как показано в следующем примере кода:

```
while (in.hasNextInt()) {
  int input = in.nextInt();
  if (n < 0) continue; // Переход к проверке in.hasNextInt()
  Обработать переменную input;
}
```

В цикле `for` оператор `continue` выполняет переход к следующему оператору обновления переменной цикла:

```
for (int i = 1; i <= target; i++) {
  int input = in.nextInt();
  if (n < 0) continue; // Переход к операции инкремента переменной i++
  Обработать переменную input;
}
```

Обычный оператор `break` прерывает только непосредственно объемлющий его цикл или оператор `switch`. Если же требуется безусловный переход в конец другого объемлющего оператора, то следует воспользоваться оператором `break` *с меткой*. Достаточно пометить оператор, из которого требуется выйти, и предоставить оператор `break` с меткой, как показано ниже. В качестве метки можно указать любое имя.

```
outer:
while (...) {
...
  while (...) {
    ...
    if (...) break outer;
      ...
  }
...
}
// Сюда происходит переход по метке
```

ВНИМАНИЕ. Метка размещается в *начале* оператора, но безусловный переход с помощью оператора **break** осуществляется в *конец* помеченного оператора.

Как пояснялось выше, обычный оператор `break` может быть использован только для выхода из цикла или оператора `switch`, тогда как оператор `break` с меткой позволяет передавать управление в конец *любого* оператора и даже блока операторов, как показано в следующем примере кода:

```
exit: {
...
if (...) break exit;
...
}
// Сюда происходит переход по метке
```

Имеется также оператор `continue` с меткой, выполняющий безусловный переход к следующему шагу помеченного цикла.

СОВЕТ. Многие программисты считают, что операторы **break** и **continue** только вносят путаницу. Пользоваться этими операторами совсем не обязательно, поскольку ту же самую логику действий можно выразить и без них. В примерах кода далее в этой книге операторы **break** и **continue** вообще не применяются.

1.7.4. Область действия локальных переменных

А теперь, когда были рассмотрены примеры вложенных блоков кода, самое время изложить правила для области действия переменных. *Локальной* называется любая переменная, объявляемая в теле метода, включая и его параметры. *Область действия* переменной представляет собой часть программы, где эта переменная доступна. Так, область действия локальной переменной простирается от места ее объявления и до конца объемлющего ее блока кода, как показано ниже.

```
while (...) {
  System.out.println(...);
  String input = in.next(); // Здесь начинается область действия
                            // переменной input
  ...
  // А здесь оканчивается область действия переменной input
}
```

Иными словами, новая копия переменной `input` создается на каждом шаге цикла, и эта переменная не существует за пределами цикла. Областью действия переменной параметра метода является все тело этого метода, как показано ниже.

```
public static void main(String[] args) { // Здесь начинается область
                                          // действия массива аргументов args
...
  // А здесь оканчивается область действия массива аргументов args
}
```

Рассмотрим пример, требующий понимания упомянутых выше правил области действия. Ниже приведен цикл, в котором подсчитывается количество попыток получить конкретное случайное число.

```
int next;
do {
  next = generator.nextInt(10);
  count++;
} while (next != target);
```

Переменную `next` пришлось объявить за пределами цикла, чтобы сделать ее доступной в условии цикла. Если бы она была объявлена в теле цикла, ее область действия простиралась бы только до конца тела цикла.

Если переменная объявляется в цикле `for`, ее область действия простирается до конца цикла, включая операторы проверки и обновления цикла, как показано ниже.

```
for (int i = 0; i < n; i++) { // Переменная i находится в области
                              // действия проверки и обновления цикла
...
}
// А здесь переменная i не определена
```

Если же значение переменной `i` требуется после цикла, ее можно объявить за пределами цикла следующим образом:

```
int i;
for (i = 0; !found && i < n; i++) {
  ...
}
// Переменная i по-прежнему доступна
```

В языке Java не допускается наличие локальных переменных с одинаковым именем в перекрывающихся областях действия. Ниже приведен характерный тому пример.

```
int i = 0;
while (...)
{
  String i = in.next(); // Ошибка переопределения переменной i!
  ...
}
```

Но если области действия не перекрываются, то переменную с одним и тем же именем можно использовать повторно, как показано ниже.

```
for (int i = 0; i < n / 2; i++) { ... }
for (int i = n / 2; i < n; i++) { ... } // Здесь переопределение
                                        // переменной i допустимо!
```

1.8. Обычные и списочные массивы

Массивы являются основополагающими конструкциями программирования, предназначенными для накопления многих элементов одного и того же типа. В языке Java имеются встроенные типы массивов, а для расширения и сокращения массивов по требованию предоставляется класс `ArrayList`. Этот класс входит в крупный каркас коллекций, рассматриваемый в главе 7.

1.8.1. Обращение с массивами

На каждый тип данных приходится соответствующий тип массива. Так, массив целочисленных значений относится к типу `int[]`, массив объектов типа `String`, т.е. символьных строк, — к типу `String[]` и т.д. В следующем примере кода объявляется переменная, способная хранить массив символьных строк:

```
String[] names;
```

Эта переменная еще не инициализирована. Чтобы инициализировать ее новым массивом, придется воспользоваться оператором `new` следующим образом:

```
names = new String[100];
```

Разумеется, оба приведенных выше оператора можно объединить, как показано ниже. Теперь переменная `names` ссылается на массив из **100** элементов, доступных в виде `names[0]` ... `names[99]`.

```
String[] names = new String[100];
```

ВНИМАНИЕ. При попытке доступа к несуществующему элементу массива, например, **names[-1]** или **names[100]**, возникает исключение **ArrayIndexOutOfBoundsException**.

Длину массива можно получить в виде свойства *array*.length. Например, в следующем цикле массив заполняется пустыми символьными строками:

```
for (int i = 0; i < names.length; i++) {
names[i] = "";
}
```

НА ЗАМЕТКУ. При объявлении переменной массива допускается пользоваться синтаксисом языка C, где после имени переменной следуют квадратные скобки **[]**, как показано ниже.

```
int numbers[];
```

Но это неудачный синтаксис, поскольку он сплетает имя **numbers** с типом **int[]**. Поэтому таким синтаксисом пользуются лишь немногие программирующие на Java.

1.8.2. Построение массива

При построении с помощью оператора new массивы заполняются значениями по умолчанию, в частности:

- массивы числового типа (включая и тип char) — нулями;
- массивы логического типа boolean — логическим значением false;
- массивы объектов — пустыми ссылками (null).

ВНИМАНИЕ. При построении массива объектов его нужно заполнить конкретными объектами. Рассмотрим следующее объявление массива:

```
BigInteger[] numbers = new BigInteger[100];
```

В данный момент объекты типа **BigInteger** пока еще отсутствуют и имеется лишь массив из 100 пустых ссылок (**null**). Их нужно заменить объектами типа **BigInteger**.

Массив можно заполнить значениями, организовав цикл, как было показано в предыдущем разделе. Но если заполняемые значения известны заранее, то их достаточно перечислить в фигурных скобках, как показано ниже. Для этого не нужен ни оператор new, ни длина массива.

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

Аналогичным синтаксисом можно воспользоваться и в том случае, если массиву требуется задать имя. Так, в следующем примере кода массив присваивается существующей переменной массива:

```
primes = new int[] { 17, 19, 23, 29, 31 };
```

НА ЗАМЕТКУ. Допускается иметь массивы нулевой длины. Такой массив можно построить как `int[0]` или `new int[] {}`. Так, если метод возвращает массив совпавших значений, среди которых отсутствуют конкретные вводимые данные, то следует возвратить массив нулевой длины. Следует, однако, иметь в виду, что это не тоже самое, что и пустое значение `null`. Так, если массив `a` имеет нулевую длину, то его свойство `a.length` равно нулю. А если массив `a` пустой (`null`), то при обращении к его свойству `a.length` возникает исключение `NullPointerException`.

1.8.3. Списочные массивы

При построении массива требуется знать его длину, а после построения массива его длина вообще не меняется. Это неудобно во многих случаях практического применения массивов. В качестве выхода из этого положения служит класс `ArrayList` из пакета `java.util`. Объект типа `ArrayList` управляет массивом внутренним образом. Когда такой массив становится слишком мал или применяется недостаточно, то автоматически создается еще один внутренний массив, в который перемещаются элементы исходного массива. Этот процесс недоступен для программиста, пользующегося списочным массивом.

Синтаксис обычных и списочных массивов совершенно разный. Для обычных массивов служит специальный синтаксис: операция `[]` для доступа к элементам массива, синтаксис ***Тип*`[]`** для массивов конкретного типа и синтаксис ***Тип*`[`*n*`]`** для построения массивов. С другой стороны, списочные массивы являются классами, и поэтому для построения их экземпляров и вызова их методов служит обычный синтаксис.

Но класс `ArrayList`, в отличие от рассмотренных ранее классов, является *обобщенным*, т.е. у него имеется параметр типа. Более подробно обобщенные классы рассматриваются в главе 6.

Для объявления переменной списочного массива служит синтаксис обобщенных классов, где тип указывается в угловых скобках, как показано ниже.

```
ArrayList<String> friends;
```

Как и для обычных массивов, в данном случае объявляется только переменная, а затем нужно построить списочный массив следующим образом:

```
friends = new ArrayList<>();
   // или new ArrayList<String>()
```

Обратите внимание на пустые угловые скобки (**<>**). Компилятор выводит параметр типа списочного массива из типа переменной. (Этот сокращенный синтаксис называется *ромбовидным*, поскольку пустые угловые скобки имеют форму ромба.)

Несмотря на то что в приведенном выше примере построения списочного массива отсутствуют аргументы, круглые скобки все равно нужно указывать в конце оператора. В итоге получается списочный массив нулевой длины. Элементы вводятся в конце списочного массива с помощью метода `add()` следующим образом:

```
friends.add("Peter");
friends.add("Paul");
```

К сожалению, для списочных массивов отсутствует синтаксис инициализации. Вводить и удалять элементы можно в любом месте списочного массива:

```
friends.remove(1);
friends.add(0, "Paul"); // вводится перед нулевым индексом
```

Для доступа к элементам списочного массива вызываются соответствующие методы вместо применения синтаксиса `[]`. Так, метод `get()` получает элемент, а метод `set()` заменяет один элемент другим, как показано ниже.

```
String first = friends.get(0);
friends.set(1, "Mary");
```

Метод `size()` получает текущую длину списочного массива. А для обхода всех элементов списочного массива организуется следующий цикл:

```
for (int i = 0; i < friends.size(); i++) {
  System.out.println(friends.get(i));
}
```

1.8.4. Классы-оболочки для примитивных типов данных

Обобщенным классам присущ следующий недостаток: в качестве параметров типа нельзя указывать примитивные типы данных. Например, выражение `ArrayList<int>` недопустимо. Для устранения этого недостатка служит *класс-оболочка*. Для каждого примитивного типа данных имеется свой класс-оболочка: `Integer`, `Byte`, `Short`, `Long`, `Character`, `Float`, `Double` и `Boolean`. Так, для накопления целочисленных значений можно воспользоваться списочным массивом `ArrayList<Integer>` следующим образом:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(42);
int first = numbers.get(0);
```

Взаимное преобразование примитивных типов и соответствующих им типов оболочек выполняется автоматически. Так, при вызове метода `add()` в приведенном выше примере кода объект типа `Integer`, хранящий значение **42**, автоматически строится в процессе так называемой *автоупаковки*.

В последней строке кода из данного примера вызывается метод `get()`, возвращающий объект типа `Integer`. Перед присваиванием переменной типа `int` этот объект подлежит так называемой *распаковке*, чтобы извлечь из него значение типа `int`.

ВНИМАНИЕ. Взаимное преобразование примитивных типов и их оболочек практически всегда прозрачно для программистов, за одним исключением. В операциях == и **!=** сравниваются ссылки на объекты, а не содержимое объектов. Так, в условии **`if (numbers.get(i) == numbers.get(j))`** не проверяется, одинаковы ли числа по индексам **`i`** и **`j`**. Как и для сравнения символьных строк, для сравнения объектов-оболочек следует вызывать метод **`equals()`**.

1.8.5. Расширенный цикл for

Нередко требуется перебрать все элементы массива. Например, в следующем примере кода вычисляется сумма всех элементов массива чисел:

```
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
  sum += numbers[i];
}
```

У этого весьма распространенного цикла имеется удобный сокращенный вариант, называемый *расширенным циклом* for:

```
int sum = 0;
for (int n : numbers) {
  sum += n;
}
```

Переменная расширенного цикла for служит для обхода всех элементов массива, а не значений индекса. В частности, переменной n присваиваются элементы массива numbers[0], numbers[1] и т.д.

Расширенный цикл for можно применять и к списочным массивам. Так, если списочный массив friends состоит из символьных строк, то все эти строки можно вывести в следующем цикле:

```
for (String name : friends) {
  System.out.println(name);
}
```

1.8.6. Копирование обычных и списочных массивов

Одну переменную массива можно скопировать в другую, как показано ниже. Но тогда обе переменные будут ссылаться на один и тот же массив (рис. 1.4).

```
int[] numbers = primes;
numbers[5] = 42; // Теперь и элемент массива primes[5] равен 42
```

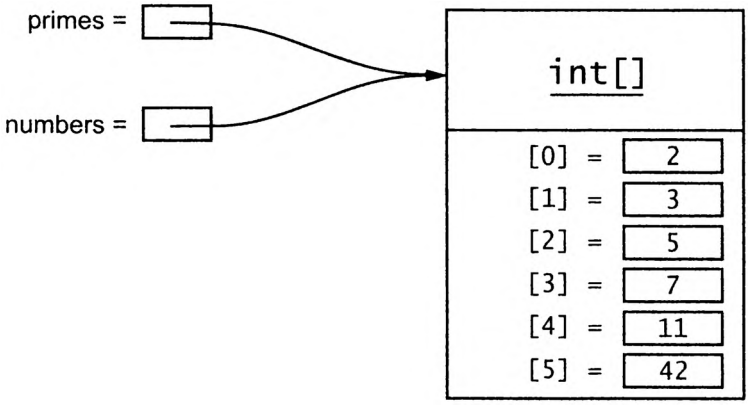


Рис. 1.4. Обе переменные ссылаются на один и тот же массив

Если такой общий доступ к массиву не требуется, то придется сделать копию массива. Для этого достаточно вызвать метод `Arrays.copyOf()` следующим образом:

```
int[] copiedPrimes = Arrays.copyOf(primes, primes.length);
```

В этом методе строится новый массив требуемой длины, и в него копируются элементы исходного массива. Аналогичным образом действуют и списочные массивы:

```
ArrayList<String> people = friends;
people.set(0, "Mary"); // Теперь и при вызове метода friends.get(0)
                       // получается символьная строка "Mary"
```

Чтобы скопировать списочный массив, достаточно построить из существующего массива новый:

```
ArrayList<String> copiedFriends = new ArrayList<>(friends);
```

С помощью конструктора можно также скопировать обычный массив в списочный массив. Для этого нужно сначала заключить массив в оболочку списочного массива, как показано ниже.

```
String[] names = ...;
ArrayList<String> friends = new ArrayList<>(Arrays.asList(names));
```

СОВЕТ. Метод **`Arrays.asList()`** может быть вызван с массивом или произвольным количеством аргументов. В последней форме его можно использовать в качестве заменителя синтаксиса инициализации, как показано ниже.

```
ArrayList<String> friends = new ArrayList<>(Arrays.asList(
            "Peter", "Paul", "Mary"));
```

Скопировать можно и списочный массив в обычный, как показано ниже. Из соображений обратной совместимости, поясняемых в главе 6, для этой цели нужно предоставить массив правильного типа.

```
String[] names = friends.toArray(new String[0]);
```

НА ЗАМЕТКУ. Выполнить взаимное преобразование массивов примитивных типов и списочных массивов соответствующих им типов оболочек не так-то просто. Например, чтобы выполнить взаимное преобразование массивов типа **`int[]`** и **`ArrayList<Integer>`**, потребуется явный цикл или поток данных **`IntStream`** (подробнее об этом речь пойдет в главе 8).

1.8.7. Алгоритмы обработки массивов

Классы `Arrays` и `Collections` предоставляют реализации общих алгоритмов для обычных и списочных массивов. Ниже показано, как заполнить обычный или списочный массив.

```
Arrays.fill(numbers, 0); // обычный массив int[]
Collections.fill(friends, ""); // списочный массив ArrayList<String>
```

Чтобы отсортировать обычный или списочный массив, достаточно воспользоваться методом `sort()` следующим образом:

```
Arrays.sort(names);
Collections.sort(friends);
```

НА ЗАМЕТКУ. Для распараллеливания сортировки обычных (но не списочных) массивов можно воспользоваться методом **`parallelSort()`**, распределяющим задание среди нескольких процессоров, если массив крупный.

Метод `Arrays.toString()` предоставляет строковое представление массива. Это особенно полезно для вывода содержимого массива в целях отладки, как показано ниже.

```
System.out.println(Arrays.toString(primes));
    // Выводит массив [2, 3, 5, 7, 11, 13]
```

У списочных массивов имеется также метод `toString()`, предоставляющий аналогичное строковое представление.

```
String elements = friends.toString();// Устанавливает строковые
                  // значения "[Peter, Paul, Mary]" в элементах массива
```

Этот метод совсем не обязательно вызывать для вывода содержимого массива в строковом представлении, поскольку об этом позаботится метод `println()`:

```
System.out.println(friends); // Вызывает метод friends.toString()
                             // автоматически и выводит результат
```

Для обработки списочных массивов имеются два полезных, приведенных ниже алгоритма, которые не имеют аналогов для обработки обычных массивов.

```
Collections.reverse(names); // Обращает элементы списочного массива
Collections.shuffle(names); // Перетасовывает элементы списочного массива
                            // в произвольном порядке
```

1.8.8. Аргументы командной строки

Как было показано ранее, у метода `main()` во всякой программе на Java имеется параметр в виде массива символьных строк, выделенный ниже полужирным.

```
public static void main(String[] args)
```

При выполнении программы в этом параметре задаются аргументы, указанные в командной строке. Рассмотрим в качестве примера следующую программу:

```
public class Greeting {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      String arg = args[i];
      if (arg.equals("-h")) arg = "Hello";
      else if (arg.equals("-g")) arg = "Goodbye";
```

```
            System.out.println(arg);
        }
    }
}
```

Если вызвать эту программу по следующей команде:

```
java Greeting -g cruel world
```

то элемент массива `args[0]` будет содержать параметр командной строки `"-g"`, элемент массива `args[1]` — слово `"cruel"`, а элемент массива `args[2]` — слово `"world"`. Обратите внимание на то, что ни команда `"java"`, ни программа `"Greeting"` методу `main()` не передается.

1.8.9. Многомерные массивы

Подлинно многомерные массивы в Java отсутствуют. Они реализуются в виде массива массивов. Так, в следующем примере кода объявляется и реализуется двухмерный массив целочисленных значений:

```
int[][] square = {
    { 16, 3, 2, 13 },
    { 3, 10, 11, 8 },
    { 9, 6, 7, 12 },
    { 4, 15, 14, 1}
};
```

Формально это одномерный массив, состоящий из массивов типа `int[]`, как показано на рис. 1.5. Для доступа к элементу такого массива служит пара квадратных скобок:

```
int element = square[1][2]; // Устанавливает значение 11 в элементе массива
```

По первому индексу выбирается массив, находящийся в строке `square[1]` двумерного массива, а по второму индексу — элемент массива из данной строки. Ряды многомерного массива можно даже менять местами, как показано ниже.

```
int[] temp = square[0];
square[0] = square[1];
square[1] = temp;
```

Если не предоставить начальное значение, то придется воспользоваться оператором `new` и указать количество строк и столбцов в многомерном массиве, как показано в следующем примере кода:

```
int[][] square = new int[4][4]; // Сначала указываются строки, а затем
                                // столбцы многомерного массива
```

Каждая строка многомерного массива внутренним образом заполняется отдельным массивом. На длину строк многомерного массива не накладывается никаких ограничений. Например, в многомерном массиве можно хранить треугольник Паскаля в следующем виде:

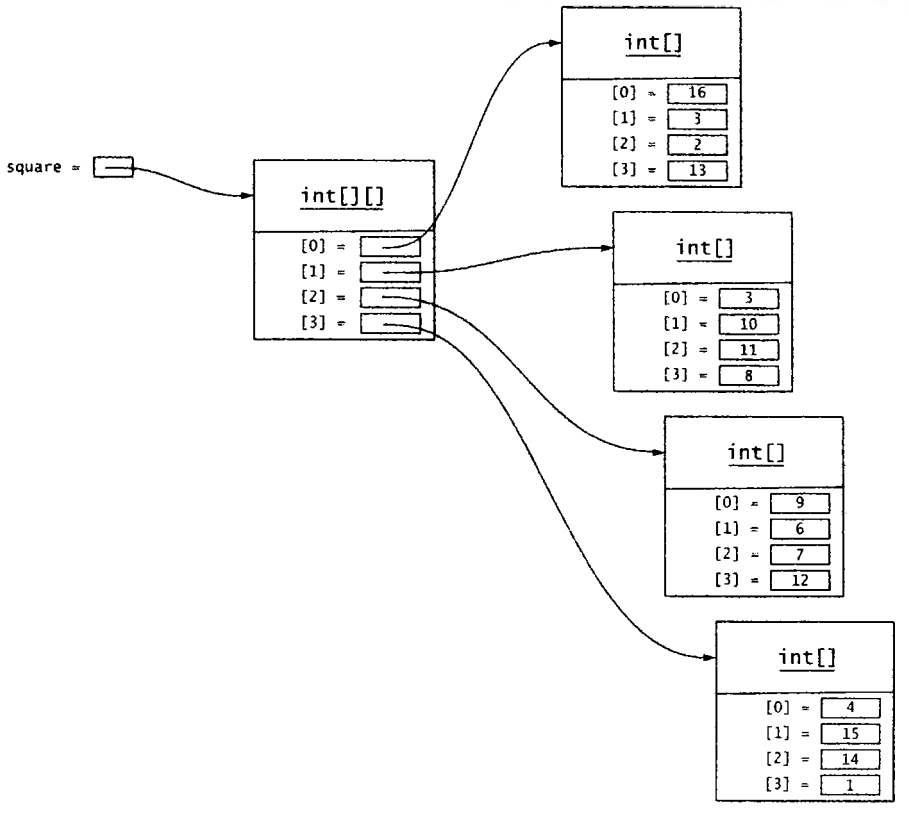


Рис. 1.5. Двумерный массив

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

Сначала строится массив из n строк следующим образом:

```
int[][] triangle = new int[n][];
```

А затем каждая строка строится и заполняется в цикле таким образом:

```
for (int i = 0; i < n; i++) {
  triangle[i] = new int[i + 1];
  triangle[i][0] = 1;
  triangle[i][i] = 1;
  for (int j = 1; j < i; j++) {
     triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
  }
}
```

Чтобы обойти весь двумерный массив, потребуются два цикла: один — для строк, а другой — для столбцов:

```
for (int r = 0; r < triangle.length; r++) {
  for (int c = 0; c < triangle[r].length; c++) {
    System.out.printf("%4d", triangle[r][c]);
  }
  System.out.println();
}
```

Для этой цели можно также воспользоваться усовершенствованными циклами for, как показано в приведенном ниже примере кода. Эти циклы пригодны как для ровных многомерных массивов, так и для неровных массивов с переменной длиной строк.

```
for (int[] row : triangle) {
  for (int element : row) {
    System.out.printf("%4d", element);
  }
  System.out.println();
}
```

СОВЕТ. Чтобы вывести списком элементы двумерного массива в целях отладки, достаточно сделать следующий вызов:

```
System.out.println(Arrays.deepToString(triangle)); // Выводит элементы
  // массива [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], ...]
```

НА ЗАМЕТКУ. Двумерные списочные массивы в Java не предусмотрены, но можно объявить переменную типа ArrayList<ArrayList<Integer>> и построить строки двумерного массива вручную.

1.9. Функциональное разложение

Если метод main() оказывается слишком длинным, программу можно разложить на несколько классов, как будет показано в главе 2. Но исходный код простых программ можно размещать в отдельных методах одного и того же класса. По причинам, которые станут понятнее в главе 2, такие методы следует объявлять с модификатором доступа static, как и сам метод main().

1.9.1. Объявление и вызов статических методов

При объявлении метода в его *заголовке* предоставляются тип возвращаемого значения (или void, если метод ничего не возвращает), имя метода, а также типы и имена параметров. А в *теле метода* предоставляется его реализация. Для возврата результата служит оператор return.

```
public static double average(double x, double y) {
  double sum = x + y;
  return sum / 2;
}
```

Объявленный таким образом метод размещается в том же самом классе, где и метод `main()`. А вызывает метод так, как показано в следующем примере кода:

```
public static void main(String[] args) {
  double a = ...;
  double b = ...;
  double result = average(a, b);
  ...
}
```

1.9.2. Массивы параметров и возвращаемые значения

Методам можно передавать массивы. В этом случае метод получает ссылку на массив, по которой он может внести изменения в массив. Так, в следующем примере кода метод меняет местами два элемента массива:

```
public static void swap(int[] values, int i, int j) {
  int temp = values[i];
  values[i] = values[j];
  values[j] = temp;
}
```

Методы могут возвращать массивы. Так, приведенный ниже метод возвращает массив, состоящий из первого и последнего значений в заданном массиве, который не видоизменяется.

```
public static int[] firstLast(int[] values) {
  if (values.length == 0) return new int[0];
  else return new int[] { values[0], values[values.length - 1] };
}
```

1.9.3. Переменное число аргументов

Некоторые методы допускают переменное число аргументов. Характерным тому примером служит упоминавшийся ранее метод `printf()`. Например, в обоих операторах,

```
System.out.printf("%d", n);
```

и

```
System.out.printf("%d %s", n, "widgets");
```

вызывается один и тот же метод `printf()`, несмотря на то, что в первом операторе этот метод вызывается с двумя аргументами, а во втором — с тремя.

В качестве примера определим метод `average()` таким образом, чтобы вызывать его с *любым* числом аргументов, например `average(3, 4.5, -5, 0)`. С этой целью

параметр переменной длины указывается с многоточием после заданного типа, как показано ниже.

```
public static double average(double... values)
```

Такой параметр, по существу, обозначает массив числовых значений типа `double`. При вызове метода `average()` этот массив создается и заполняется аргументами. А в теле метода он используется аналогично любому другому массиву, как показано ниже.

```
public static double average(double... values) {
  double sum = 0;
  for (double v : values) sum += v;
  return values.length == 0 ? 0 : sum / values.length;
}
```

Теперь этот метод можно вызвать следующим образом:

```
double avg = average(3, 4.5, 10, 0);
```

Если в массиве уже имеются аргументы, то распаковывать их не нужно. Вместо списка аргумент можно передать в массив следующим образом:

```
double[] scores = { 3, 4.5, 10, 0 };
double avg = average(scores);
```

Параметр с переменным числом аргументов должен быть указан *последним* в объявлении метода, хотя ему могут предшествовать другие параметры. Например, в определении следующего метода гарантируется наличие хотя бы одного аргумента:

```
public static double max(double first, double... rest) {
  double result = first;
  for (double v : rest) result = Math.max(v, result);
  return result;
}
```

Упражнения

1. Напишите программу, вводящую целочисленное значение и выводящую его в двоичной, восьмеричной и шестнадцатеричной форме. Организуйте вывод обратного значения в виде шестнадцатеричного числа с плавающей точкой.
2. Напишите программу, вводящую целочисленное (как положительное, так и отрицательное) значение угла и нормализующую его в пределах от `0` до `359` градусов. Попробуйте сделать это сначала с помощью операции `%`, а затем метода `floorMod()`.
3. Напишите программу, вводящую три целочисленных значения и выводящую самое большое из них, используя только условную операцию. Сделайте то же самое с помощью метода `Math.max()`.

4. Напишите программу, выводящую наименьшее и наибольшее положительные значения типа `double`. *Подсказка:* воспользуйтесь методом `Math.nextUp()` из прикладного программного интерфейса Java API.
5. Что произойдет, если привести числовое значение типа `double` к значению типа `int`, которое больше самого большого значения типа `int`? Попробуйте сделать это.
6. Напишите программу, вычисляющую факториал $n! = 1 \times 2 \times \ldots \times n$, используя класс `BigInteger`. Вычислите факториал числа **1000**.
7. Напишите программу, вводящую два числа в пределах от **0** до **65535**, сохраняющую их в переменных типа `short` и вычисляющую их сумму, разность, произведение, частное и остаток без знака, не преобразуя эти величины в тип `int`.
8. Напишите программу, вводящую символьную строку и выводящую все ее непустые подстроки.
9. В разделе 1.5.3 был приведен пример сравнения двух символьных строк `s` и `t` при вызове метода `s.equals(t)`, но не с помощью операции `s != t`. Придумайте другой пример, в котором не применяется метод `substring()`.
10. Напишите программу, составляющую произвольную символьную строку из букв и цифр, генерируя произвольное значение типа `long` и выводя его по основанию **36**.
11. Напишите программу, вводящую текстовую строку и выводящую все символы, не представленные в коде ASCII, вместе с их значениями в Юникоде.
12. В состав комплекта разработки Java Development Kit входит архивный файл `src.zip` с исходным кодом библиотеки Java. Разархивируйте этот файл и с помощью избранного вами инструментального средства для поиска текста найдите в этом исходном коде примеры применения последовательностей операторов `break` и `continue` с меткой. Выберите один из этих примеров и перепишите его без оператора с меткой.
13. Напишите программу, выбирающую и выводящую лотерейную комбинацию из шести отдельных чисел в пределах от **1** до **49**. Чтобы выбрать шесть отдельных чисел, начните со списочного массива, заполняемого числами от **1** до **49**. Выберите произвольный индекс и удалите элемент массива. Повторите эти действия шесть раз подряд. Выведите полученный результат в отсортированном порядке.
14. Напишите программу, вводящую двумерный массив целочисленных значений и определяющую, содержится ли в нем магический квадрат (т.е. одинаковая сумма значений во всех строках, столбцах и диагоналях). Принимая строки вводимых данных, разбивайте их на отдельные целочисленные значения, прекратив этот процесс, когда пользователь введет пустую строку. Например, на следующие вводимые данные:

```
16 3 2 13
3 10 11 8
```

```
9 6 7 12
4 15 14 1
(Пустая строка)
```

программа должна ответить утвердительно.

15. Напишите программу, сохраняющую треугольник Паскаля вплоть до заданной величины `n` в переменной типа `ArrayList<ArrayList<Integer>>`.
16. Усовершенствуйте упоминавшийся ранее метод `average()` таким образом, чтобы он вызывался хотя бы с одним параметром.

Глава 2

Объектно-ориентированное программирование

В этой главе...

В объектно-ориентированном программировании (ООП) все внимание сосредоточено на взаимодействии объектов, поведение которых определяется теми классами, к которым они принадлежат. Java относится к числу первых основных языков программирования, где полностью реализованы принципы ООП. Как было показано в главе 1, каждый метод в Java объявляется в классе и каждое значение, кроме значений примитивных типов, представлено объектом. Из этой главы вы узнаете, каким образом реализуются классы и методы.

Основные положения этой главы приведены ниже.

1. Модифицирующие методы изменяют состояние объекта, а методы доступа не изменяют его.
2. В языке Java переменные содержат не объекты, а ссылки на них.
3. Переменные экземпляра и реализации методов объявляются в определении класса.
4. Метод экземпляра вызывается для объекта, доступного по ссылке `this`.
5. Конструктор имеет такое же имя, как и класс. У класса может быть несколько (перегружаемых) конструкторов.
6. Статические переменные относятся к любым объектам. А статические методы не вызываются для объектов.
7. Классы организованы в пакеты. Поэтому следует пользоваться объявлениями импорта, чтобы не указывать постоянно имя пакета в программах.
8. Одни классы могут быть вложены в другие классы.
9. Внутренний класс является нестатическим вложенным классом. У его экземпляров имеется ссылка на объект объемлющего класса, который его построил.
10. Утилита **`javadoc`** обрабатывает исходные файлы, производя из них HTML-файлы с объявлениями и комментариями, которыми программисты снабжают исходный код.

2.1. Обращение с объектами

В прежние времена, еще до изобретения объектов, программы писали с вызовом *функций*. Когда функция вызывается, она возвращает результат, которым можно пользоваться, не беспокоясь о том, как он был вычислен. У функций имеется важное преимущество: они позволяют разделять работу. В частности, можно вызвать функцию, написанную кем-то другим, даже не зная, как она решает возложенную на нее задачу.

Объекты вносят еще одно измерение. У каждого объекта может быть свое *состояние*, которое оказывает влияние на результаты, получаемые из вызова метода. Так, если создать объект `in` класса `Scanner` и сделать вызов `in.next()`, то объект сохранит то, что было прочитано прежде, и выдаст следующий маркер ввода.

Если используются объекты, реализованные кем-то другим и для них вызываются методы, то совсем не обязательно знать, что именно происходит под спудом. Этот принцип называется инкапсуляцией и является основополагающим в ООП.

В какой-то момент у вас может возникнуть потребность сделать свою работу доступной для других программистов, предоставив им объекты, которыми они могли бы пользоваться. Этой цели в Java служит *класс* — механизм создания и применения объектов с одинаковым поведением.

Рассмотрим типичную задачу манипулирования календарными датами. Календари не упорядочены в том отношении, что они состоят из месяцев разной продолжительности и високосных лет, не говоря уже о корректировочных секундах. Поэтому имеет смысл привлечь специалистов, разбирающихся в особенностях составления календарей и способных предоставить реализации, которыми могли бы воспользоваться другие программисты. И для этой цели естественным образом подходят объекты. В частности, дата может быть представлена объектом, методы которого способны предоставить сведения, например, о том, на какой день недели приходится конкретная дата или какая дата наступит завтра.

Специалисты, разбирающиеся в особенностях вычисления дат, предоставили классы, написанные на Java для дат и других связанных с ними понятий вроде недель. Если требуется произвести расчеты по датам, то для этой цели можно воспользоваться соответствующим классом, чтобы создать объекты дат и вызвать для них методы, например, метод, возвращающий день недели или следующую дату.

Вряд ли вам захочется вникать в подробности арифметических операций с датами, если вы специализируетесь на чем-нибудь другом. А для того чтобы другие программисты могли воспользоваться плодами ваших знаний и трудов, можете предоставить им соответствующие классы. И даже если вы не собираетесь делиться плодами ваших знаний и трудов с другими программистами, в своей работе вам все равно будет удобнее пользоваться классами, чтобы структурировать свои программы согласованным образом. Но прежде чем перейти к разъяснению особенностей объявления классов, рассмотрим нетривиальный пример применения объектов.

По команде **cal** в Unix на заданный месяц и указанный год выводится календарь в формате, аналогичном следующему:

```
Mon Tue Wed Thu Fri Sat Sun
                            1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26  27  28  29
30
```

Как же реализовать такую команду на Java? В стандартной библиотеке Java имеется класс `LocalDate`, предназначенный для выражения даты в некотором неозначенном месте. В данном случае требуется объект этого класса, представляющий первый день месяца. Ниже показано, как получить его.

```
LocalDate date = LocalDate.of(year, month, 1);
```

Чтобы перейти к следующей дате, достаточно сделать вызов `date.plusDays(1)`. В итоге будет построен новый объект типа `LocalDate` со следующей датой. В рас-

сматриваемом здесь примере прикладной задачи достаточно еще раз присвоить результат переменной date следующим образом:

```
date = date.plusDays(1);
```

Для получения сведений о дате, например, месяце, на который она приходится, вызываются методы, выделенные ниже полужирным. Эти сведения требуются для того, чтобы выводить календарь в пределах заданного месяца.

```
while (date.getMonthValue() == month) {
  System.out.printf("%4d", date.getDayOfMonth());
  date = date.plusDays(1);
  ...
}
```

Еще один метод возвращает день недели, на который приходится дата:

```
DayOfWeek weekday = date.getDayOfWeek();
```

При этом возвращается объект другого класса, называемого DayOfWeek. Чтобы рассчитать отступ для первого дня месяца в календаре, нужно знать числовое значение соответствующего дня недели. Для этой цели вызывается следующий метод:

```
int value = weekday.getValue();
for (int i = 1; i < value; i++)
  System.out.print("   ");
```

Метод getValue() придерживается внутреннего соглашения о том, что выходные дни приходятся на конец недели. Следовательно, он возвращает значение 1, обозначающее понедельник, значение 2, обозначающее вторник, и так далее до значения 7, обозначающего воскресенье.

НА ЗАМЕТКУ. Вызовы методов можно *связывать в цепочку*, как в следующем примере кода:

```
int value = date.getDayOfWeek().getValue();
```

В данном примере вызов первого метода применяется к объекту **date** и возвращается объект типа **DayOfWeek**. А затем для этого объекта вызывается метод **getValue()**.

Весь исходный код рассматриваемого здесь примера программы можно найти среди примеров исходного кода, прилагаемых к данной книге и доступных по ссылке, приведенной в ее введении. Решить задачу вывода календаря оказалось нетрудно потому, что разработчики класса LocalDate предоставили в наше распоряжение ряд полезных методов. Далее в этой главе будет показано, как реализовать методы для своих классов.

2.1.1. Методы доступа и модифицирующие методы

Рассмотрим вызов метода date.plusDays(1) еще раз. Разработчики класса LocalDate могли бы реализовать метод plusDays(). В частности, они могли бы внести изменения в состояние объекта date, не возвращая результат. А с другой стороны,

они могли бы оставить объект date без изменений и возвратить вновь построенный объект типа LocalDate. Как видите, они выбрали второй способ.

Если метод изменяет объект, для которого он вызывается, то такой метод называется *модифицирующим*. А если вызываемый метод оставляет объект без изменения, то он называется *методом доступа*. В частности, метод plusDays() из класса LocalDate является методом доступа.

На самом деле *все* методы из класса LocalDate являются методами доступа. Такая ситуация становится все более типичной, поскольку модификация может оказаться рискованной, особенно в том случае, если два вычисления видоизменяют объект одновременно. В настоящее время большинство компьютеров оснащено несколькими процессорными блоками, и поэтому безопасный параллельный доступ к ним представляет серьезное затруднение. Это затруднение можно, в частности, разрешить, сделав объекты *неизменяемыми* и снабдив их только методами доступа.

Тем не менее во многих случаях модификация объектов желательна. Характерным примером модифицирующего метода служит метод add() из класса ArrayList. После вызова метода add() объект списочного массива изменяется, как показано ниже.

```
ArrayList<String> friends = new ArrayList<>();
    // Списочный массив friends пуст
friends.add("Peter");
    // Списочный массив friends имеет длину 1
```

2.1.2. Ссылки на объекты

В некоторых языках программирования (например, C++) переменная может содержать объект, т.е. биты, образующие состояние объекта. Совсем иначе дело обстоит в языке Java, где переменная может хранить только *ссылку* на объект. Сам же объект может храниться в каком-нибудь другом месте, тогда как ссылка служит для его обнаружения способом, зависящим от конкретной реализации (рис. 2.1).

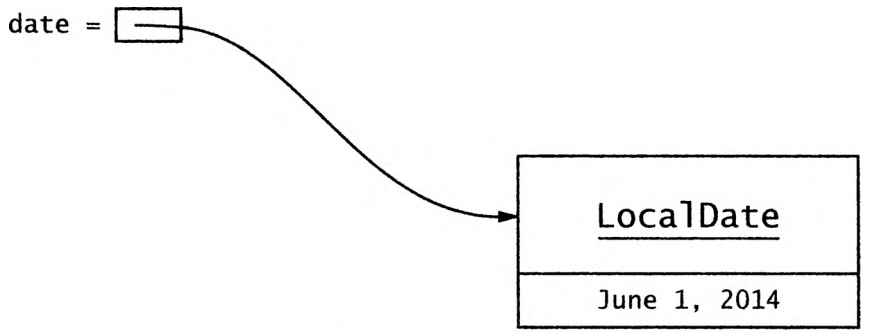


Рис. 2.1. Ссылка на объект

НА ЗАМЕТКУ. Ссылки на объекты в Java ведут себя как указатели в C и C++, за исключением того, что они совершенно безопасны. Если в языках C и C++ можно видоизменять указатели, применяя их для перезаписи произвольных ячеек памяти, то в языке Java по ссылке может быть доступен только конкретный объект.

Когда переменной, хранящей ссылку на один объект, присваивается ссылка на другой объект, получаются две ссылки на один и тот же объект, как показано ниже.

```
ArrayList<String> people = friends;
  // Теперь переменные people и friends ссылаются на один и тот же объект
```

Если видоизменить совместно используемый объект, то его модификация будет доступна по обеим ссылкам. Рассмотрим следующий пример вызова:

```
people.add("Paul");
```

Теперь списочный массив `people` имеет длину **2**, как, впрочем, и списочный массив `friends` (рис. 2.2). Если рассуждать формально, то это не совсем верно. Ведь списочный массив `people` не является объектом. Это ссылка на объект, а по существу, на списочный массив длиной **2**.

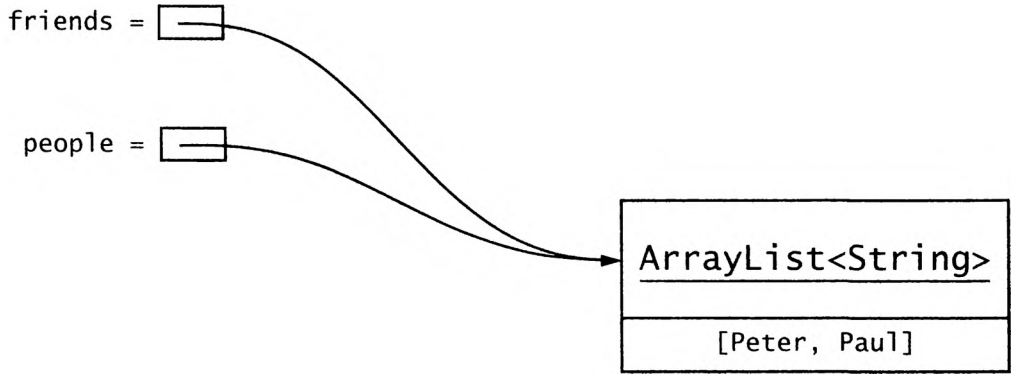


Рис. 2.2. Две ссылки на один и тот же объект

Чаще всего такое совместное использование объектов оказывается эффективным и удобным. Следует, однако, иметь в виду, что совместно используемый объект может быть видоизменен по любой ссылке на него.

Но если в классе отсутствуют модифицирующие методы (примером тому служит класс `String` или `LocalDate`), то и беспокоиться особенно не о чем. Объект такого класса нельзя изменить, и поэтому ссылки на него можно предоставлять свободно.

Объектная переменная может вообще не ссылаться ни на один из объектов, если установить в ней пустое значение `null`, как показано ниже. Это может быть удобно, если для переменной `date` пока еще отсутствует объект, на который она может ссылаться, или же если требуется указать на особый случай, например, неизвестную дату.

```
LocalDate date = null; // Теперь переменная date не ссылается
                       // ни на один из объектов
```

ВНИМАНИЕ. Пустые значения могут представлять известную опасность, когда они неожиданны. Так, вызов метода по пустой ссылке (`null`) приводит к исключению типа **`NullPointerException`**, которое на самом деле должно называться **`NullReferenceException`**. Именно по этой причине

и не рекомендуется пользоваться пустыми значениями **null** как необязательными. Вместо них лучше пользоваться типом **Optional**, который мы рассмотрим в главе 8.

И наконец, проанализируем еще раз следующие операции присваивания объектов переменным:

```
date = LocalDate.of(year, month, 1);
date = date.plusDays(1);
```

После первого присваивания переменная `date` ссылается на первый день месяца. В результате вызова метода `plusDays()` возвращается новый объект типа `LocalDate`, а после второго присваивания переменная `date` ссылается на новый объект. Что же происходит с первым объектом?

Ссылка на первый объект отсутствует, и поэтому он больше не требуется. В конечном итоге *сборщик "мусора"* восстановит оперативную память, выделенную для первого объекта, чтобы она стала доступной для дальнейшего применения. В языке Java этот процесс полностью автоматизирован, и поэтому программистам вообще не нужно беспокоиться об освобождении оперативной памяти, занимаемой удаляемыми объектами.

2.2. Реализация классов

А теперь обратимся к примеру реализации собственного класса. Чтобы продемонстрировать различные языковые правила, создадим класс `Employee`. Этот класс представляет работника, имеющего имя и зарплату. В данном примере имя работника не подлежит изменению, тогда как его зарплата может вырасти, как часто бывает при повышении по службе.

2.2.1. Переменные экземпляра

Из описания объектов работников можно выяснить, что состояние такого объекта обозначается следующими двумя значениями: имя и зарплата. В языке Java для описания состояния объекта служит *переменная экземпляра*. Такие переменные объявляются в классе, как показано ниже. Таким образом, каждый объект или *экземпляр* класса `Employee` содержит две переменные экземпляра.

```
public class Employee {
  private String name;
  private double salary;
  ...
}
```

В языке Java переменные экземпляра обычно объявляются закрытыми (`private`). Это означает, что они доступны только методам из того же самого класса. Для такой защиты переменных экземпляра имеются две причины. Во-первых, требуется выбрать те части прикладной программы, в которых допускается видоизменять переменные, а во-вторых, изменить внутреннее представление в любой момент. Например, сведения

о работниках требуется сохранить в базе данных, а в представляющем их объекте — только первичный ключ. Если методы снова реализованы в классе таким образом, чтобы действовать как и прежде, то пользователям данного класса совсем не важно, как это сделано.

2.2.2. Заголовки методов

А теперь перейдем к реализации методов в классе `Employee`. При объявлении метода указывается его имя, типы и имена его параметров, а также возвращаемый тип, как в следующей строке кода:

```
public void raiseSalary(double byPercent)
```

Этот метод принимает параметр типа `double`, но не возвращает никакого значения, о чем свидетельствует возвращаемый тип `void`. А метод `getName()` имеет совсем другую сигнатуру, как показано ниже. У этого метода отсутствуют параметры, но он возвращает объект типа `String`.

```
public String getName()
```

НА ЗАМЕТКУ. Как правило, методы объявляются открытыми (**`public`**), а это означает, что всякий может вызвать такой метод. Иногда вспомогательные методы объявляются закрытыми (**`private`**), чтобы их применение ограничивалось пределами того класса, в котором они объявлены. Подобным образом объявляются те методы, которые не имеют отношения к пользователям класса, особенно если они зависят от подробностей реализации. Закрытые методы можно благополучно изменить или удалить, если изменяется реализация.

2.2.3. Тела методов

В заголовке следующего метода предоставляется его тело:

```
public void raiseSalary(double byPercent) {
  double raise = salary * byPercent / 100;
  salary += raise;
}
```

Если метод возвращает значение, то для его возврата служит ключевое слово `return`:

```
public String getName() {
  return name;
}
```

Объявления методов размещаются в объявлении класса следующим образом:

```
public class Employee {
  private String name;
  private double salary;

  public void raiseSalary(double byPercent) {
    double raise = salary * byPercent / 100;
```

```
    salary += raise;
  }

  public String getName() {
    return name;
  }
  ...
}
```

2.2.4. Вызов методов экземпляра

Рассмотрим следующий пример вызова метода:

```
fred.raiseSalary(5);
```

В этом вызове аргумент **5** служит для инициализации переменной параметра `byPercent`, что равнозначно следующей операции присваивания:

```
double byPercent = 5;
```

Затем в данном методе выполняются следующие действия:

```
double raise = fred.salary * byPercent / 100;
fred.salary += raise;
```

Обратите внимание на то, что переменная `salary` применяется к экземпляру, для которого вызывается данный метод. В отличие от методов, представленных в конце предыдущей главы, метод наподобие `raiseSalary()` оперирует экземпляром класса. Поэтому такой метод называется *методом экземпляра*. Все методы в Java, которые не объявлены как статические (`static`), являются методами экземпляра.

Как видите, методу `raiseSalary()` передаются следующие два значения: ссылка на объект, для которого он вызывается, а также аргумент, с которым он вызывается. Формально и то и другое считается параметром метода, но в Java, как и в других языках объектно-ориентированного программирования, ссылка на объект играет особую роль. Ее иногда еще называют *получателем* вызова метода.

2.2.5. Ссылка `this`

Когда метод вызывается для объекта, этот объект устанавливается в ссылке `this`. При желании ссылку `this` можно, хотя и не обязательно, указывать в реализации метода, как показано ниже.

```
public void raiseSalary(double byPercent) {
  double raise = this.salary * byPercent / 100;
  this.salary += raise;
}
```

Некоторые предпочитают именно такой стиль программирования, поскольку он позволяет провести явное различие между локальными переменными и переменными экземпляра. Так, из приведенного выше примера кода становится очевидно, что `raise` — это локальная переменная, а `salary` — переменная экземпляра.

Нередко ссылка this употребляется в том случае, если нет желания присваивать другие имена переменным параметров метода, как показано в следующем примере кода:

```
public void setSalary(double salary) {
  this.salary = salary;
}
```

Когда переменная экземпляра и локальная переменная имеют одинаковое имя, то не уточненное имя (например, salary) обозначает локальную переменную, тогда как ссылка this.salary — переменную экземпляра.

НА ЗАМЕТКУ. В некоторых языках программирования переменные экземпляра декорируются особым образом, как, например, **_name** и **_salary**. Это вполне допустимо и в Java, хотя обычно не делается.

НА ЗАМЕТКУ. Если требуется, ссылку **this** можно даже объявить как параметр метода (но не конструктора) следующим образом:

```
public void setSalary(Employee this, double salary) {
  this.salary = salary;
}
```

Но такой синтаксис употребляется крайне редко. Другой способ явно указать получателя метода состоит в применении аннотации, как поясняется в главе 11.

2.2.6. Вызов по значению

Когда объект передается методу, последний получает копию ссылки на объект. По этой ссылке можно получить доступ или видоизменить объект параметра, как в следующем примере кода:

```
public class EvilManager {
  private Random generator;
  ...
  public void giveRandomRaise(Employee e) {
    double percentage = 10 * generator.nextGaussian();
    e.raiseSalary(percentage);
  }
}
```

Рассмотрим следующий вызов:

```
boss.giveRandomRaise(fred);
```

Ссылка fred копируется в переменную параметра e (рис. 2.3). Метод видоизменяет объект, разделяемый обеими ссылками.

В языке Java вообще нельзя написать метод, обновляющий параметры примитивного типа. В частности, попытка увеличить в методе значение типа double не пройдет, как показано ниже.

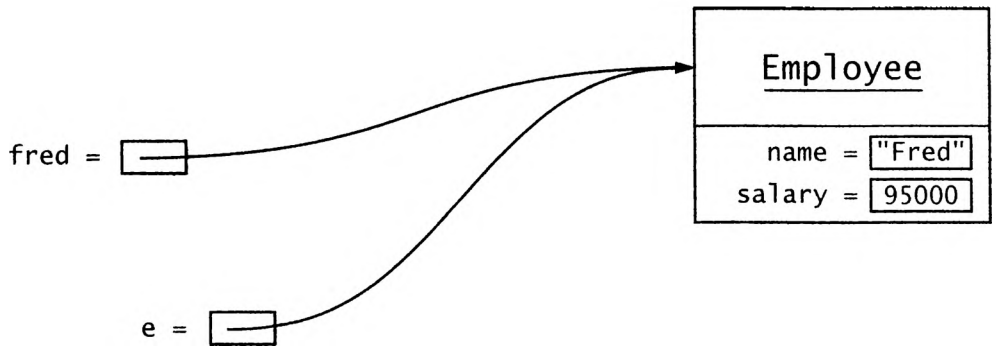


Рис. 2.3. Переменная параметра содержит копию ссылки на объект

```
public void increaseRandomly(double x) { // Не пройдет!
  double amount = x * generator.nextDouble();
  x += amount;
}
```

Если сделать следующий вызов:

```
boss.increaseRandomly(sales);
```

то аргумент `sales` будет скопирован в параметр `x`. Затем значение параметра `x` будет увеличено, но это не изменит значение аргумента `sales`. Ведь переменная параметра выйдет из области своего действия, а увеличение ее значения не даст никакого полезного результата.

По той же самой причине нельзя написать метод, заменяющий ссылку на объект чем-то другим. Например, следующий метод будет действовать не так, как предполагалось:

```
public class EvilManager {
  ...
  public void replaceWithZombie(Employee e) {
    e = new Employee("", 0);
  }
}
```

При вызове

```
boss.replaceWithZombie(fred);
```

ссылка `fred` копируется в переменную `e`, в которой затем устанавливается другая ссылка. При возврате из данного метода переменная `e` выходит из области своего действия, а в итоге ссылка `fred` остается прежней.

НА ЗАМЕТКУ. Некоторые считают, что в Java применяется вызов объектов по ссылке. Но это не так, о чем свидетельствует последний пример. В тех языках программирования, где поддерживается вызов по ссылке, в теле метода можно изменить содержимое передаваемой ему переменной. А в языке Java все параметры (ссылки на объекты и значения примитивных типов) передаются *только* по значению.

2.3. Построение объектов

Для завершения класса осталось сделать еще один шаг. Для этого нужно предоставить конструктор, как подробно поясняется в последующих разделах.

2.3.1. Реализация конструкторов

Конструктор объявляется таким же образом, как и метод. Но имя конструктора совпадает с именем класса, а кроме того, у него отсутствует *возвращаемый тип.*

```
public Employee(String name, double salary) {
  this.name = name;
  this.salary = salary;
}
```

НА ЗАМЕТКУ. Приведенный выше конструктор является открытым. Но иногда полезно иметь и закрытые конструкторы. Например, в упоминавшемся ранее классе **LocalDate** отсутствуют открытые конструкторы. Вместо этого пользователи данного класса получают объекты из так называемых "фабричных" методов вроде **now()** и **of()**. Эти методы вызывают закрытый конструктор.

ВНИМАНИЕ. Если случайно указать возвращаемый тип, как в следующем объявлении конструктора:

```
void public void Employee(String name, double salary)
```

то тем самым будет объявлен метод **Employee()**, а не конструктор!

Конструктор выполняется, когда применяется оператор new. Например, в выражении

```
new Employee("James Bond", 500000)
```

назначается объект класса Employee и вызывается тело конструктора, где в переменных экземпляра устанавливаются значения аргументов, предоставляемых конструктору.

Оператор new возвращает ссылку на построенный объект. Эта ссылка обычно сохраняется в переменной следующим образом:

```
Employee james = new Employee("James Bond", 500000);
```

или передается методу таким образом:

```
ArrayList<Employee> staff = new ArrayList<>();
staff.add(new Employee("James Bond", 500000));
```

2.3.2. Перегрузка

Имеется возможность предоставить не одну версию конструктора. Так, если требуется упростить моделирование безымянных рабочих пчелок, достаточно предоставить второй конструктор, принимающий только зарплату в качестве параметра.

```
public Employee(double salary) {
  this.name = "";
  this.salary = salary;
}
```

Теперь в классе Employee имеются два конструктора. Вызов каждого из них зависит от передаваемых аргументов, как в приведенном ниже примере кода. В таком случае говорят, что конструктор *перегружается*.

```
Employee james = new Employee("James Bond", 500000);
    // вызывается конструктор Employee(String, double)
Employee anonymous = new Employee(40000);
    // вызывается конструктор Employee(double)
```

НА ЗАМЕТКУ. Метод перегружается, если имеется несколько его версий с одним и тем же именем, но с разными параметрами. Например, имеются перегружаемые версии метода **println()** с параметрами типа **int**, **double**, **String** и т.д. Конструкторы зачастую перегружаются, поскольку им нельзя присвоить имя.

2.3.3. Вызов одного конструктора из другого

Если имеется несколько конструкторов, они, как правило, выполняют некоторую общую работу. Поэтому во избежание дублирования кода лучше разместить общую инициализацию в каком-нибудь одном конструкторе.

Один конструктор можно вызвать из другого, но только в *первом* операторе тела конструктора. При таком вызове вместо имени конструктора указывается ключевое слово this.

```
public Employee(double salary) {
  this("", salary); // Вызывается конструктор Employee(String, double)
  // Далее могут следовать остальные операторы
}
```

НА ЗАМЕТКУ. В данном случае ключевое слово **this** *не* обозначает ссылку на создаваемый объект, оно обозначает специальный синтаксис, употребляемый только для вызова одного конструктора из другого конструктора того же самого класса.

2.3.4. Инициализация по умолчанию

Если не установить переменную экземпляра в конструкторе явным образом, то в ней автоматически устанавливается значение по умолчанию: нуль — для числовых типов; логическое значение false — для типа boolean; пустое значение null — для ссылок на объекты. Например, в качестве параметра конструктору можно передать имя неоплачиваемого стажера следующим образом:

```
public Employee(String name) {
  // в переменной экземпляра salary автоматически
  // устанавливается нулевое значение
```

```
  this.name = name;
}
```

НА ЗАМЕТКУ. В этом отношении переменные экземпляра заметно отличаются от локальных переменных. Напомним, что локальные переменные следует инициализировать явным образом.

Числовые значения зачастую удобнее инициализировать нулем. Но ссылки на объекты нередко служат источниками ошибок. Допустим, что в следующем конструкторе `Employee(double)` не установлена пустая символьная строка в переменной экземпляра `name`:

```
public Employee(double salary) {
  // в переменной экземпляра автоматически
  // устанавливается пустое значение null
  this.salary = salary;
}
```

Если кто-нибудь попытается вызвать метод `getName()`, то в конечном итоге получит пустую ссылку, которая явно не предполагалась. Вычисление условного выражения вроде следующего:

```
if (e.getName().equals("James Bond"))
```

приведет к исключению, возникающему при обнаружении пустого указателя.

2.3.5. Инициализация переменных экземпляра

Для любой переменной экземпляра можно указать начальное значение, как в следующем примере кода:

```
public class Employee {
  private String name = "";
  ...
}
```

Подобная инициализация происходит после назначения объекта и перед выполнением конструктора. Следовательно, исходное значение имеется во всех конструкторах. Разумеется, в некоторых из них оно может быть переопределено.

Помимо инициализации переменной экземпляра при ее объявлении, в объявление класса можно также включить произвольные *блоки инициализации*, как показано ниже.

```
public class Employee() {
  private String name = "";
  private int id;
  private double salary;

  { // Блок инициализации
    Random generator = new Random();
    id = 1 + generator.nextInt(1_000_000);
  }
```

```
  public Employee(String name, double salary) {
    ...
  }
}
```

НА ЗАМЕТКУ. Блоками инициализации обычно не пользуются. Большинство программирующих на Java размещают длинный код инициализации в теле вспомогательного метода и вызывают его из конструкторов.

Инициализация переменных экземпляра и блоки инициализации выполняются в том порядке, в каком они появляются в объявлении класса, но *перед* телом конструктора.

2.3.6. Конечные переменные экземпляра

Переменную экземпляра можно объявить как конечную (`final`). Такая переменная должна быть непременно инициализирована к концу определения каждого конструктора. А после этого конечную переменную уже нельзя будет видоизменить снова. Например, переменная экземпляра `name` может быть объявлена в классе `Employee` как `final`, поскольку она вообще не изменяется после построения объекта. Ведь в этом классе отсутствует метод `setName()`. Ниже показано, как это делается.

```
public class Employee {
  private final String name;
  ...
}
```

НА ЗАМЕТКУ. Если модификатор **`final`** доступа применяется к ссылке на изменяемый объект, то он лишь обозначает, что изменению не подлежит только ссылка. А изменять объект вполне допустимо, как показано ниже.

```
public class Person {
  private final ArrayList<Person> friends = new ArrayList<>();
    // Вводить элементы в этот списочный массив вполне допустимо
...
}
```

Методы могут изменять содержимое списочного массива, на который ссылается переменная **`friends`**, но они не могут заменить эту ссылку на него другой ссылкой. В частности, она вообще не может стать пустой (**`null`**).

2.3.7. Конструкторы без аргументов

Во многих классах имеется конструктор без аргументов, создающий объект, состояние которого соответственно устанавливается по умолчанию. В качестве примера ниже приведен конструктор без аргументов для класса `Employee`.

```
public Employee() {
name = "";
salary = 0;
}
```

Подобно тому, как ответчику, неспособному покрыть расходы на защиту своих интересов в суде, предоставляется государственный защитник, класс без конструкторов автоматически снабжается конструктором без аргументов, который практически ничего не делает. Все переменные экземпляра остаются со своими значениями по умолчанию (нулем, логическим значением `false` или пустым значением `null`), если только они не инициализируются явным образом. Следовательно, у каждого класса имеется по крайней мере один конструктор.

НА ЗАМЕТКУ. Если у класса уже имеется конструктор, то он *не* получает автоматически еще один конструктор без аргументов. Если же требуется предоставить такой конструктор, его придется написать самостоятельно.

НА ЗАМЕТКУ. В предыдущих разделах было показано, что именно происходит при построении объекта. В некоторых языках программирования, особенно в C++, принято указывать, что именно должно происходить при удалении объекта. А в Java предусмотрен механизм "полного завершения" существования объекта, когда он утилизируется из оперативной памяти сборщиком "мусора". Но этот механизм действует в непредсказуемые моменты времени, и поэтому пользоваться им не следует. В главе 5 будет представлен другой механизм высвобождения ресурсов, в том числе закрытия файлов.

2.4. Статические переменные и методы

Во всех приведенных ранее примерах программ метод `main()` помечался модификатором доступа `static`. В последующих разделах поясняется назначение этого модификатора доступа.

2.4.1. Статические переменные

Если объявить переменную в классе как `static`, то она должна быть единственной в классе. С другой стороны, у каждого объекта имеется своя копия переменной экземпляра. Допустим, каждому работнику требуется присвоить отдельный идентификационный номер, как показано ниже. Это даст возможность обмениваться последним присвоенным ему идентификационным номером.

```
public class Employee {
  private static int lastId = 0;
  private int id;
  ...
  public Employee() {
    lastId++;
    id = lastId;
  }
}
```

У каждого объекта типа `Employee` имеется своя переменная экземпляра `id`, но лишь одна переменная `lastId` принадлежит самому классу, а не любому конкретному экземпляру класса. При построении нового объекта типа `Employee` общая переменная `lastId` инкрементируется, и ее значение устанавливается в переменной

экземпляра id. Следовательно, каждому работнику присваивается отдельный идентификационный номер в виде значения переменной экземпляра id.

ВНИМАНИЕ. Приведенный выше фрагмент кода окажется неработоспособным, если объекты типа **Employee** могут быть построены параллельно в нескольких потоках исполнения. В главе 10 будет показано, как исправить это положение.

НА ЗАМЕТКУ. В связи с изложенным выше возникает следующий вопрос: почему переменная, принадлежащая классу, а не отдельным его экземплярам, называется "статической"? Этот термин является бессмысленным пережитком, унаследованным от языка C++, в котором он позаимствован как ключевое слово из языка C без всякой связи, вместо того чтобы придумать что-нибудь более подходящее. Более описательным был бы термин "переменная класса".

2.4.2. Статические константы

Изменяемые статические переменные употребляются редко, тогда как статические константы (т.е. переменные, объявляемые как `static final`) — очень часто. Например, в классе `Math` объявляется статическая константа `PI`, доступная в прикладном коде по ссылке `Math.PI`:

```
public class Math {
  ...
  public static final double PI = 3.14159265358979323846;
  ...
}
```

Без ключевого слова `static` константа `PI` была бы просто переменной экземпляра класса `Math`. Это означало бы, что для доступа к переменной экземпляра `PI` потребовался бы объект данного класса, причем у каждого такого объекта имелась бы своя копия переменной экземпляра `PI`.

Ниже приведен пример статической константы, содержащей объект, а не число. Зачастую расточительно и небезопасно строить новый генератор случайных чисел всякий раз, когда требуется случайное число. Много лучше иметь генератор случайных чисел, общий для всех экземпляров класса.

```
public class Employee {
  private static final Random generator = new Random();
  private int id;
  ...
  public Employee() {
    id = 1 + generator.nextInt(1_000_000);
  }
}
```

Еще одним примером статической константы служит стандартный поток вывода `System.out`. Эта константа объявляется в классе `System` следующим образом:

```
public class System {
  public static final PrintStream out;
  ...
}
```

ВНИМАНИЕ. Несмотря на то что статическая константа `out` объявляется в классе `System` как `final`, имеется метод `setOut()` для установки в константе `System.out` другого потока вывода. Это платформенно-ориентированный метод, не реализуемый в Java и способный обойти механизмы управления доступом в языке Java. Такая необычная ситуация встречалась в первых версиях Java, но вряд ли возникает теперь.

2.4.3. Статические блоки инициализации

В предыдущих разделах статические переменные инициализировались при их объявлении. Но иногда им требуется дополнительная инициализация. В таком случае ее можно заключить в *статический блок инициализации*, как показано ниже.

```
public class CreditCardForm {
  private static final ArrayList<Integer>
          expirationYear = new ArrayList<>();
  static {
    // добавить следующие двадцать лет в списочный массив
    int year = LocalDate.now().getYear();
    for (int i = year; i <= year + 20; i++) {
      expirationYear.add(i);
    }
  }
  ...
}
```

Статическая инициализация происходит при первоначальной загрузке класса. Как и переменные экземпляра, статические переменные принимают по умолчанию значение `0`, `false` или `null`, если только не установить в них другое значение явным образом. Все виды инициализации статических переменных и статические блоки инициализации выполняются в том порядке, в каком они появляются в объявлении класса.

2.4.4. Статические методы

Статическими называют методы, которые не оперируют объектами. Например, метод `pow()` из класса `Math` является статическим. В следующем выражении:

```
Math.pow(x, a)
```

вычисляется степень x^a. Для выполнения данной математической операции никакого объекта типа `Math` не требуется.

В главе 1 уже демонстрировался статический метод, объявляемый с модификатором доступа `static` следующим образом:

```
public class Math {
  public static double pow(double base, double exponent) {
    ...
  }
}
```

Почему бы не сделать статический метод `pow()` методом экземпляра? Он не может быть методом экземпляра типа `double`, поскольку это примитивный тип, а

примитивные типы в Java не являются классами. Его можно было бы сделать методом экземпляра класса Math, но тогда пришлось бы построить объект типа Math, чтобы вызвать этот метод.

Еще одной распространенной причиной для существования статических классов служит снабжение чужого класса дополнительными функциональными возможностями. Например, было бы неплохо иметь метод, возвращающий случайное целочисленное значение в заданных пределах? С этой целью нельзя ввести метод в класс Random из стандартной библиотеки, но можно предоставить статический метод следующим образом:

```
public class RandomNumbers {
  public static int nextInt(Random generator, int low, int high) {
    return low + generator.nextInt(high - low + 1);
  }
}
```

Этот метод вызывается следующим образом:

```
int dieToss = RandomNumbers.nextInt(gen, 1, 6);
```

НА ЗАМЕТКУ. Статический метод разрешается вызывать для объекта. Например, вместо вызова **LocalDate.now()** для получения текущей даты можно сделать вызов **date.now()** для объекта **date** из класса **LocalDate**, хотя и это не имеет никакого смысла. Ведь метод **now()** не обращает внимания на объект **date** для вычисления результата. Многие программирующие на Java посчитали бы такой стиль программирования неудачным.

Статические методы не оперируют объектами, и поэтому переменные экземпляра недоступны из статического метода. Тем не менее статические методы могут получить доступ к статическим переменным в своем классе. Например, в методе RandomNumbers.nextInt() генератор случайных чисел можно присвоить статической переменной следующим образом:

```
public class RandomNumbers {
  private static Random generator = new Random();
  public static int nextInt(int low, int high) {
    return low + generator.nextInt(high - low + 1);
      // Статическая переменная generator вполне доступна
  }
}
```

2.4.5. Фабричные методы

Чаще всего статические методы употребляются как *фабричные*, т.е. как статические методы, возвращающие новые экземпляры класса. Например, в классе NumberFormat применяются статические методы, возвращающие форматирующие объекты для различных стилей оформления, как показано ниже.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
```

```
System.out.println(currencyFormatter.format(x)); // Выводит $0.10
System.out.println(percentFormatter.format(x)); // Выводит 10%
```

А почему бы не воспользоваться вместо этого конструктором? Два конструктора могут отличаться только типами их параметров. А наличие двух конструкторов без аргументов не допускается.

Более того, конструктор `new NumberFormat(...)` выдает объект класса `NumberFormat`, а фабричный метод возвращает объект его подкласса. В действительности приведенные выше фабричные методы возвращают экземпляры класса `DecimalFormat`. (Подробнее о подклассах речь пойдет в главе 4.)

Фабричный метод может также возвращать совместно используемый объект вместо излишнего построения новых объектов. Например, в результате вызова метода `Collections.emptyList()` возвращается совместно используемый неизменяемый пустой список.

2.5. Пакеты

В языке Java связанные вместе классы размещаются в одном пакете. Пакеты удобны для упорядочения результатов трудов одних программистов и отделения их от библиотек исходного кода других программистов. Как было показано выше, стандартная библиотека Java распространяется в виде целого ряда пакетов, включая следующие: `java.lang`, `java.util`, `java.math` и т.д.

Основной причиной употребления пакетов является гарантия однозначности имен классов. Допустим, двум программистам пришла в голову одна и та же блестящая мысль предоставить класс `Element`. (На самом деле эта мысль посетила по крайней мере пять разработчиков прикладного программного интерфейса Java API.)

Если эти программисты разместят свои классы в разных пакетах, то никакого конфликта имен не возникнет. В последующих разделах будет разъяснено, как обращаться с пакетами.

2.5.1. Объявления пакетов

Имя пакета представляет собой разделяемый точками список идентификаторов, например `java.util.regex`. Чтобы гарантировать однозначность имен пакетов, рекомендуется обозначать их именами доменов Интернета (которые также являются однозначными), но только в обратном порядке. Например, у меня имеется доменное имя `horstmann.com`. Для своих проектов я пользуюсь именами пакетов вроде `com.horstmann.corejava`. Единственным важным исключением из этого правила является стандартная библиотека Java, где имена пакетов начинаются с обозначения `java` или `javax`.

ПРИМЕЧАНИЕ. В языке Java вложение пакетов не предусмотрено. Например, пакеты **`java.util`** и **`java.util.regex`** никак не связаны друг с другом. Каждый из них является независимой коллекцией классов.

Чтобы разместить класс в пакете, достаточно ввести оператор `package` в первой строке исходного файла, как показано ниже. Теперь класс `Employee` находится в пакете `com.horstmann.corejava`, а его *полностью уточненное имя* таково: `com.horstmann.corejava.Employee`.

```
package com.horstmann.corejava;

public class Employee {
   ...
}
```

Имеется также *пакет по умолчанию* без имени, который можно использовать для простых программ. Чтобы ввести класс в пакет по умолчанию, указывать оператор `package` не нужно. Тем не менее пользоваться пакетом по умолчанию не рекомендуется.

Когда файлы классов извлекаются из файловой системы, имя пути к файлу должно совпадать с именем пакета. Например, файл `Employee.class` должен находиться в подкаталоге `com/horstmann/corejava`.

Если упорядочить исходные файлы аналогичным образом и скомпилировать их из каталога, содержащего первоначальные имена пакетов, то файлы классов будут автоматически размещены в нужном месте. Допустим, в классе `EmployeeDemo` используются объекты типа `Employee` и он компилируется по следующей команде:

```
javac com/horstmann/corejava/EmployeeDemo.java
```

По этой команде компилятор сформирует файлы классов `com/horstmann/corejava/EmployeeDemo.class` и `com/horstmann/corejava/Employee.class`. А для выполнения программы придется указать полностью уточненное имя класса следующим образом:

```
java com.horstmann.corejava.EmployeeDemo
```

ВНИМАНИЕ. Если исходный файл не находится в подкаталоге, совпадающем с именем его пакета, то компилятор `javac` не выдаст никаких предупреждений и сформирует файл класса. Но тогда этот файл придется разместить в нужном месте, хотя сделать это будет не так-то просто (см. упражнение 12 в конце главы).

СОВЕТ. Компилятор `javac` рекомендуется выполнять с параметром `-d`. В таком случае файлы классов формируются в отдельном каталоге, не загромождая дерево каталогов с исходным кодом, которое будет иметь правильную структуру подкаталогов.

2.5.2. Путь к классу

Вместо того чтобы хранить файлы классов в файловой системе, их можно разместить в одном или нескольких архивных файлах, называемых JAR-файлами. Такой архив можно создать с помощью утилиты `jar`, входящей в комплект JDK, как показано

ниже. Параметры командной строки этой утилиты аналогичны параметрам утилиты **tar** в ОС Unix. Подобная операция обычно выполняется над библиотеками пакетов.

```
jar cvf library.jar com/mycompany/*.class
```

НА ЗАМЕТКУ. По умолчанию для архивирования исходного кода в JAR-файлах употребляется формат ZIP. Имеется также возможность выбрать другую схему сжатия данных, называемую pack200 и предназначенную для более эффективного сжатия файлов классов.

СОВЕТ. Архивные JAR-файлы можно использовать для упаковки не только программ, но и библиотек. JAR-файл формируется по следующей команде:

```
jar cvfe program.jar com.mycompany.MainClass com/mycompany/*.class
```

И тогда программа выполняется по такой команде:

```
java -jar program.jar
```

Если в проекте используются архивные JAR-файлы библиотек, то компилятору и виртуальной машине нужно сообщить, где находятся эти файлы, указав *путь к классам*. Путь к классам может содержать следующие составляющие:

- Каталоги, содержащие файлы классов (в подкаталогах, совпадающих с именами их пакетов).
- Архивные JAR-файлы.
- Каталоги, содержащие архивные JAR-файлы.

Утилиты **javac** и **java** имеют параметр **-classpath**, который можно сократить до **-cp**. Ниже приведен пример вызова утилиты **java** с параметром **-classpath**, обозначающим путь к классам. Этот путь к классам состоит из следующих трех элементов: текущего каталога (**.**) и двух архивных JAR-файлов в каталоге **../libs**.

```
java -classpath .:../libs/lib1.jar:../libs/lib2.jar com.mycompany.MainClass
```

НА ЗАМЕТКУ. В Windows для разделения элементов пути вместо двоеточий ставят точку с запятой:

```
java -classpath .;..\libs\lib1.jar;..\libs\lib2.jar com.mycompany.
MainClass
```

Если имеется много архивных JAR-файлов, то их следует разместить в одном каталоге и включить в выполняемую команду с помощью метасимвола подстановки следующим образом:

```
java -classpath .:../libs/\* com.mycompany.MainClass
```

НА ЗАМЕТКУ. В Unix метасимвол в каталоге * должен быть экранирован во избежание раскрытия команд в командном процессоре.

ВНИМАНИЕ. Компилятор **javac** всегда ищет файлы в текущем каталоге, а утилита **java** делает это только в том случае, если текущий каталог (.) явно указан в пути к классам. Если же путь к классам не установлен, то ничего страшного, поскольку путь к классам по умолчанию содержит текущий каталог (.). Но если установить путь к классам вручную и при этом забыть указать в нем текущий каталог (.), то программы не будут выполняться, хотя и скомпилируются без ошибок.

Для установки пути к классам предпочтительнее применять параметр **-classpath**. С другой стороны, путь к классам можно установить в переменной окружения CLASSPATH, хотя это зависит от конкретного командного процессора. Так, в командном процессоре **bash** путь к классам устанавливается в переменной окружения CLASSPATH следующим образом:

```
export CLASSPATH=.:/home/username/project/libs/\*
```

а в Windows таким образом:

```
SET CLASSPATH=.;C:\Users\username\project\libs\*
```

ВНИМАНИЕ. Установить переменную окружения **CLASSPATH** можно глобально (например, в файле **.bashrc** или на панели управления Windows). Но многие программисты жалеют об этом, когда забывают глобальные настройки и недоумевают, почему их классы не найдены.

ВНИМАНИЕ. Некоторые рекомендуют вообще обходиться без пути к классам, размещая все архивные JAR-файлы в специальном каталоге **jre/lib/ext**, куда виртуальная машина обращается в поисках установленных расширений. На самом деле это очень плохой совет. Прежде всего, код, загружающий классы вручную, работает не правильно, если он размещен в каталоге для расширений. Более того, программисты еще меньше будут помнить о малоизвестном им каталоге **jre/lib/ext**, чем о переменной окружения **CLASSPATH**, пребывая в полном недоумении, когда загрузчик классов проигнорирует тщательно установленный ими путь к классам, загружая давно забытые классы из каталога для расширений.

2.5.3. Область действия пакетов

В приведенных ранее примерах кода уже встречались модификаторы доступа public и private. Языковые средства, помеченные модификатором доступа public, могут использоваться в любым классе, а языковые средства, помеченные модификатором доступа private, — только в том классе, в котором они объявлены. Если же не указать ни один из модификаторов доступа public или private, то языковое средство (т.е. класс, метод или переменная) может быть доступно всем методам из одного и того же пакета.

Область действия пакета удобна для доступа к служебным классам и методам, которые требуются методам из пакета, но не интересуют пользователей этого пакета. Нередко область действия пакета применяется и для тестирования. В частности, тестовые классы можно разместить в том же самом пакете, чтобы они имели доступ к содержимому тестируемых классов.

НА ЗАМЕТКУ. Исходный файл может состоять из нескольких классов, но хотя бы один из них вполне может быть объявлен как **public**. Если исходный файл содержит открытый класс, то имя этого файла должно совпадать с именем данного класса.

Область действия пакета не подходит для переменных как устанавливаемая по умолчанию. Типичная ошибка программистов состоит в том, что они забывают указать модификатор доступа `private` и неумышленно делают переменную экземпляра доступной для всего пакета. Ниже приведен характерный тому пример из класса `Window`, входящего в состав пакета `java.awt`.

```
public class Window extends Container {
  String warningString;
  ...
}
```

Переменная `warningString` не является закрытой и поэтому доступна методам из всех классов в пакете `java.awt`. На самом деле она не требуется ни одному из методов, кроме тех, которые определены в самом классе `Window`. По-видимому, программист просто забыл объявить ее с модификатором доступа `private`.

Но это может быть связано с нарушением безопасности, поскольку пакеты открыты для расширения. Любой класс можно внедрить в пакет, предоставив соответствующий оператор `package`. Разработчики Java перестраховались от подобной атаки злоумышленников, соответственно оснастив загрузчик классов (`ClassLoader`), который не будет загружать ни один из классов, полностью уточненные имена которых начинаются с **java**.

Если вам потребуется аналогичная защита ваших собственных пакетов, их придется разместить в *герметичном* архивном JAR-файле, а также предоставить так называемый *манифест* — текстовый файл, содержимое которого аналогично следующему:

```
Name: com/mycompany/util/
Sealed: true
Name: com/mycompany/misc/
Sealed: true
```

Затем следует выполнить команду **jar**, аналогичную приведенной ниже.

```
jar cvfm library.jar manifest.txt com/mycompany/*/*.class
```

2.5.4. Импорт классов

Оператор `import` дает возможность пользоваться классами, не указывая их полностью уточненные имена. Так, если в прикладном коде используется следующая строка кода:

```
import java.util.Random;
```

то вместо полностью уточненного имени класса `java.util.Random` далее в прикладном коде можно указать имя `Random`.

НА ЗАМЕТКУ. Объявлять импорт классов удобно, но не обязательно. Все объявления импорта классов можно опустить и пользоваться полностью уточненными их именами в любом месте прикладного кода.

```
java.util.Random generator = new java.util.Random();
```

Операторы `import` размещаются перед первым объявлением класса в исходном файле, но ниже оператора `package`. Все классы можно импортировать из пакета, используя метасимвол подстановки следующим образом:

```
import java.util.*;
```

Используя метасимвол подстановки, можно импортировать только классы, но не пакеты. Так, для получения всех пакетов, имена которых начинаются с **java**, нельзя воспользоваться оператором

```
import java.*;
```

При импорте нескольких пакетов вполне возможен конфликт имен. Например, в обоих пакетах, `java.util` и `java.sql`, содержится класс `Date`. Допустим, в программе требуется импортировать оба этих пакета следующим образом:

```
import java.util.*;
import java.sql.*;
```

Если класс `Date` не применяется в программе, то никаких осложнений не возникает. Но если в ней происходит обращение к классу `Date` без указания имени пакета, то компилятор выдаст соответствующее предупреждение. В таком случае требующийся класс можно импортировать, как показано ниже. А если в программе требуются оба класса с одинаковым именем из разных пакетов, то при обращении хотя бы к одному из них следует указать его полностью уточненное имя.

```
import java.util.*;
import java.sql.*;
import java.sql.Date;
```

НА ЗАМЕТКУ. Оператор **`import`** служит для удобства программирования, а в файлах классов все имена классов полностью уточнены.

НА ЗАМЕТКУ. Оператор **`import`** в Java совсем не похож на директиву **`#include`** в C и C++. Эта директива включает в себя заголовочные файлы для компиляции. А импорт файлов не приводит к их компиляции и лишь сокращает имена классов подобно оператору **`using`** в C++.

2.5.5. Статический импорт

Имеется отдельная форма оператора `import`, позволяющая импортировать статические методы и переменные. Так, если ввести в начале исходного файла следующую строку кода:

```
import static java.lang.Math.*;
```

то далее в прикладном коде можно воспользоваться статическими методами и переменными из класса Math, не указывая имя этого класса в качестве префикса.

```
sqrt(pow(x, 2) + pow(y, 2)) // т.е. Math.sqrt, Math.pow
```

Имеется также возможность импортировать конкретный статический метод или переменную следующим образом:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

НА ЗАМЕТКУ. Как будет показано в главах 3 и 8, объявления статического импорта **java.util.Comparator** и **java.util.stream.Collectors** часто применяются для доступа к интерфейсу **Comparator** и классу **Collectors**, поскольку они предоставляют большое количество статических методов.

ВНИМАНИЕ. Статические методы, переменные или поля нельзя импортировать из класса, входящего в пакет по умолчанию.

2.6. Вложенные классы

В предыдущем разделе было показано, как организовать классы в пакеты. С другой стороны, один класс можно разместить в другом. Такой класс называется *вложенным*. Подобное размещение классов может оказаться удобным для ограничения их доступности или во избежание загромождения пакета типичными именами вроде Element, Node или Item. В языке Java имеются две разновидности вложенных классов с несколько отличающимся поведением. Рассмотрим их по очереди в последующих разделах.

2.6.1. Статические вложенные классы

Рассмотрим в качестве примера класс Invoice для выписки счета-фактуры на товары, каждый из которых имеет описание, количество и цену за штуку. Эти товары можно определить в классе Item, вложенном в класс Invoice:

```
public class Invoice {
  private static class Item { // Класс Item вложен в класс Invoice
    String description;
    int quantity;
    double unitPrice;

    double price() { return quantity * unitPrice; }
  }

  private ArrayList<Item> items = new ArrayList<>();
  ...
}
```

Отложим до следующего раздела пояснение причин, по которым внутренний класс Item объявлен как static, а до тех пор будем воспринимать это как должное. В определении класса Item нет ничего особенного, кроме управления доступом к нему. Этот класс закрыт в пределах класса Invoice, и поэтому он доступен только методам из класса Invoice. Именно по этой причине переменные экземпляра во внутреннем классе не объявлены закрытыми.

Ниже приведен пример метода, строящего объекты внутреннего класса.

```
public class Invoice {
  ...
  public void addItem(String description, int quantity,
                      double unitPrice) {
    Item newItem = new Item();
    newItem.description = description;
    newItem.quantity = quantity;
    newItem.unitPrice = unitPrice;
    items.add(newItem);
  }
}
```

Один класс, вложенный в другой, можно сделать открытым. В этом случае придется воспользоваться обычным механизмом инкапсуляции.

```
public class Invoice {
  public static class Item { // Открытый вложенный класс
    private String description;
    private int quantity;
    private double unitPrice;

    public Item(String description, int quantity, double unitPrice) {
      this.description = description;
      this.quantity = quantity;
      this.unitPrice = unitPrice;
    }
    public double price() { return quantity * unitPrice; }
      ...
    }

  private ArrayList<Item> items = new ArrayList<>();

  public void add(Item item) { items.add(item); }
  ...
}
```

Теперь всякий может построить объекты типа Item, указав полностью уточненное имя Invoice.Item следующим образом:

```
Invoice.Item newItem = new Invoice.Item("Blackwell Toaster", 2, 19.95);
myInvoice.add(newItem);
```

По существу, этот вложенный класс Invoice.Item мало чем отличается от класса InvoiceItem, объявляемого за пределами любого другого класса. Вложение класса лишь делает очевидным тот факт, что класс Item представляет товары в выписываемом счете-фактуре.

2.6.2. Внутренние классы

В предыдущем разделе был продемонстрирован пример вложенного класса, объявленного как `static`. А в этом разделе поясняется, что произойдет, если опустить модификатор доступа `static` в объявлении вложенного класса. Такой класс называется *внутренним*.

Рассмотрим в качестве следующего примера социальную сеть, где у каждого ее члена имеются друзья, также являющиеся ее членами:

```
public class Network {
  public class Member { // Класс Member является внутренним
                        // для класса Network
    private String name;
    private ArrayList<Member> friends;

    public Member(String name) {
      this.name = name;
      friends = new ArrayList<>();
    }
    ...
  }

  private ArrayList<Member> members;
...
}
```

Внутренний класс существенно отличается от вложенного, если в его объявлении опущен модификатор доступа `static`. В частности, члену, представленному объектом типа `Member`, известно, к какой социальной сети он принадлежит. Рассмотрим, как это происходит. Ниже представлен метод для ввода члена в социальную сеть.

```
public class Network {
  ...
  public Member enroll(String name) {
    Member newMember = new Member(name);
    members.add(newMember);
    return newMember;
  }
}
```

Пока что ничего особенного, по-видимому, не происходит. Члена можно ввести в социальную сеть и получить ссылку на него, как показано ниже.

```
Network myFace = new Network();
Network.Member fred = myFace.enroll("fred");
```

А теперь допустим, член Фред считает, что данная социальная сеть его больше не устраивает, и он хочет покинуть ее. Для этого нужно сделать следующий вызов:

```
fred.leave();
```

Ниже приведена реализация метода `leave()`.

```
public class Network {
  public class Member {
    ...
    public void leave() {
      members.remove(this);
    }
  }

  private ArrayList<Member> members;
  ...
}
```

Как видите, методу из внутреннего класса могут быть доступны переменные экземпляра из его внешнего класса. В этом случае они являются переменными экземпляра создавшего их объекта myFace внешнего класса, представляющего теперь уже неинтересную для данного члена социальную сеть.

Именно этим внутренний класс отличается от статического вложенного класса. У каждого объекта внутреннего класса имеется ссылка на объект объемлющего его класса. Например, вызов метода

```
members.remove(this);
```

фактически равнозначен такому вызову:

```
outer.members.remove(this);
```

где **outer** — скрытая ссылка на объемлющий класс.

Такая ссылка отсутствует в статическом вложенном классе подобно тому, как у статического метода отсутствует ссылка this. Таким образом, статический вложенный класс следует применять в том случае, если экземплярам вложенного класса не нужно знать, к какому именно экземпляру объемлющего класса они принадлежат, а внутренний класс — в том случае, если подобная информация важна.

Во внутреннем классе можно также вызывать методы из внешнего класса по ссылке на экземпляр последнего. Допустим, что во внешнем классе имеется метод для снятия члена с регистрации в социальной сети. В таком случае этот метод можно вызвать в методе leave() следующим образом:

```
public class Network {
  public class Member {
    ...
    public void leave() {
      unenroll(this);
    }
  }

  private ArrayList<Member> members;

  public Member enroll(String name) { ... }
  public void unenroll(Member m) { ... }
  ...
}
```

В этом случае вызов

```
unenroll(this);
```

фактически равнозначен такому вызову:

```
outer.unenroll(this);
```

где **outer** — скрытая ссылка на объемлющий класс.

2.6.3. Правила специального синтаксиса для внутренних классов

Как пояснялось в предыдущем разделе, ссылка на внешний класс из объекта внутреннего класса обозначается как **outer**. На самом же деле синтаксис внешних ссылок немного сложнее. В частности, выражение ***ВнешнийКласс*.this** обозначает ссылку на внешний класс. Например, метод `leave()` из внутреннего класса можно написать следующим образом:

```
public void leave() {
  Network.this.members.remove(this);
}
```

В данном случае синтаксис `Network.this` не нужен. Простое обращение по ссылке `members` неявно подразумевает употребление ссылки на внешний класс. Но иногда ссылку на внешний класс требуется указывать явно. В качестве примера ниже приведен метод для проверки принадлежности члена к конкретной социальной сети.

```
public class Network {
  public class Member {
    ...
    public boolean belongsTo(Network n) {
      return Network.this == n;
    }
  }
}
```

При построении объекта внутреннего класса в нем запоминается объект объемлющего класса, который его построил. В предыдущем разделе новый член социальной сети создавался с помощью метода

```
public class Network {
  ...
  Member enroll(String name) {
    Member newMember = new Member(name);
    ...
  }
}
```

Это сокращенный вариант следующего выражения:

```
Member newMember = this.new Member(name);
```

Конструктор внутреннего класса можно вызвать для любого экземпляра внешнего класса следующим образом:

```
Network.Member wilma = myFace.new Member("Wilma");
```

НА ЗАМЕТКУ. Во внутренних классах нельзя объявлять статические члены, кроме компилируемых констант, чтобы исключить неоднозначность толкования термина "статический". Но значит ли это, что в виртуальной машине имеется лишь один экземпляр статического члена, приходящийся на каждый внешний объект? Разработчики языка Java оставили этот вопрос открытым.

НА ЗАМЕТКУ. Исторически сложилось так, что внутренние классы были внедрены в язык Java в тот момент, когда спецификация виртуальной машины считалась завершенной. Таким образом, они преобразуются в обычные классы со скрытыми ссылками из переменных экземпляра на экземпляры объемлющего класса. В упражнении 14, приведенном в конце главы, предлагается исследовать подобное преобразование.

НА ЗАМЕТКУ. Еще одной разновидностью внутренних классов являются *локальные классы*, подробнее рассматриваемые в главе 3.

2.7. Документирующие комментарии

В комплект JDK входит очень полезная утилита **javadoc**, автоматически составляющая в формате HTML документацию из исходных файлов. В действительности документация на прикладной программный интерфейс Java API, оперативно доступная по ссылке, приведенной в главе 1, является результатом применения утилиты **javadoc** к исходному коду стандартной библиотеки Java.

Если ввести в исходный код комментарии, начинающиеся с разделителя **/****, то на исходный код можно легко составить документацию профессионального вида. Это очень удобно, поскольку дает возможность хранить исходный код и документацию на него в одном месте. В прошлом программистам нередко приходилось размещать документацию в отдельном файле, и поэтому появление расхождений в исходном коде и комментариях к нему было лишь вопросом времени. Если же документирующие комментарии находятся в том же самом файле, где и исходный код, то обновить и то и другое не составит большого труда, выполнив снова утилиту **javadoc**.

2.7.1. Ввод комментариев

Утилита **javadoc** извлекает из исходного кода сведения о следующих элементах.

- Пакеты.
- Открытые классы и интерфейсы.
- Открытые и защищенные переменные.
- Открытые и защищенные конструкторы и методы.

Интерфейсы представлены в главе 3, а защищенные языковые средства — в главе 4. Каждое из этих языковых средств можно (и должно) снабдить комментариями. Каждый комментарий размещается перед описываемым в нем языковым средством. Комментарий начинается с разделителя `/**` и завершается разделителем `*/`.

Каждый документирующий комментарий `/** ... */` состоит из текста в свободной форме с последующими дескрипторами. А каждый дескриптор начинается со знака `@`, например `@author` или `@param`. *Первое предложение* в тексте документирующего комментария должно быть кратким заявлением о назначении комментируемого языкового средства. Утилита **javadoc** автоматически формирует сводные страницы документации, извлекая эти предложения из документирующих комментариев.

В тексте документирующего комментария можно употреблять такие модификаторы HTML-разметки, как `<em>...</em>` — для выделения текста; `<code>...</code>` — для оформления текста моноширинным шрифтом; `<strong>...</strong>` — для выделения текста полужирным и даже `<img ...>` — для включения в текст изображений. Однако следует избегать употребления дескрипторов заголовков `<h`*n*`>` или правил `<hr>`, поскольку они могут помешать форматированию документации.

НА ЗАМЕТКУ. Если комментарии содержат ссылки на другие файлы, например, изображений (диаграмм или элементов пользовательского интерфейса), эти файлы следует разместить в подкаталоге, входящем в каталог **doc-files** с исходными файлами. Утилита **javadoc** скопирует содержимое каталога **doc-files** в каталог с документацией. Для этого каталог **doc-files** нужно указать в ссылке, например, следующим образом:

```
<img src="doc-files/uml.png" alt="UML diagram"/>
```

2.7.2. Комментарии к классам

Комментарии к классу должны размещаться непосредственно перед его объявлением. Автора (или авторов) документации и ее версию можно задокументировать с помощью дескрипторов `@author` и `@version`. Ниже приведен пример комментария к классу.

```
/**
* объект <code> класса Invoice</code> представляет счет-фактуру
* с позициями заказа для каждой его части
* @author Fred Flintstone
* @author Barney Rubble
* @version 1.1
*/
public class Invoice {
  ...
}
```

НА ЗАМЕТКУ. Указывать знак * в начале каждой строки комментария совсем не обязательно, Но в большинстве ИСР это делается автоматически, а в некоторых из них знаки * даже переупорядочиваются при изменении разрывов строк.

2.7.3. Комментарии к методам

Комментарии к каждому методу размещаются непосредственно перед его объявлением. Документированию подлежат следующие языковые средства.

- Каждый параметр с комментарием `@param` *описание переменной*.
- Возвращаемое значение, если оно не типа `void`: `@return` *описание*.
- Любые генерируемые исключения (см. главу 5): `@throws` *описание класса исключения*.

Ниже приведен пример комментария к методу.

```
/**
* Поднимает зарплату работника.
* @param byPercent проценты, на которые поднимается зарплата
*                  (например, числовое значение 10 означает 10%)
* @return сумма увеличения зарплаты
*/
public double raiseSalary(double byPercent) {
  double raise = salary * byPercent / 100;
  salary += raise;
  return raise;
}
```

2.7.4. Комментарии к переменным

Документировать следует лишь открытые переменные. Как правило, это относится к статическим константам. Ниже приведен пример комментария к переменной, объявляемой как статическая константа.

```
/**
* Количество дней в году на Земле, исключая високосные годы
*/
public static final int DAYS_PER_YEAR = 365;
```

2.7.5. Общие комментарии

Во всех документирующих комментариях можно пользоваться дескриптором `@since` для описания версии, в которой данное средство стало доступным, как показано следующем примере:

```
@since version 1.7.1
```

Дескриптор `@deprecated` вводит комментарии о том, что соответствующий класс, метод или переменная больше не применяется. В тексте такого комментария должна быть предложена замена не рекомендованного к употреблению языкового средства. Например:

```
@deprecated Использовать <code>метод setVisible(true)</code> вместо данного
```

НА ЗАМЕТКУ. Имеется также аннотация `@Deprecated`, используемая компилятором для выдачи предупреждений, когда применяются не рекомендованные к употреблению языковые средства (см. главу 11). У аннотации отсутствует механизм для предложения замены не рекомендованного к употреблению языкового средства, и поэтому такое средство должно быть снабжено как аннотацией, так и документирующим комментарием.

2.7.6. Ссылки

В соответствующие части документации, составляемой с помощью утилиты **javadoc**, или во внешнюю документацию можно ввести гиперссылки, используя дескрипторы `@see` и `@link`. В частности, дескриптор `@see` вводит гиперссылку в разделе документации "См. также". Его можно употреблять как в классах, так и в методах. Ниже приведены возможные формы ссылки.

- *пакет.класс#метка_средства*
- `<a href="...">`*метка*`</a>`
- "*текст*"

Первая форма ссылки наиболее употребительна. В ней указывается имя класса, метода или переменной, а утилита **javadoc** автоматически вставляет гиперссылку в документацию, Так, в следующем примере кода:

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

делается ссылка на метод `raiseSalary(double)` из класса `com.horstmann.corejava.Employee`. Имя пакета или его имя вместе с именем класса можно опустить. В таком случае поиск указанного в ссылке языкового средства будет осуществлен в текущем пакете или классе.

Однако в данной форме ссылки нужно ставить знак **#**, а не точку, чтобы отделить имя класса от имени метода или переменной. Компилятор Java достаточно развит логически, чтобы отличать разное назначение знака точки в качестве разделителя имен пакетов, подпакетов, обычных и внутренних классов, их методов и переменных. Но утилита **javadoc** не настолько развита логически и поэтому требует помощи от пользователя в различении имен.

Если за дескриптором `@see` следует знак **<**, это означает, что указана гиперссылка. Ссылку можно сделать по какому угодно URL. Например:

```
@see <a href="http://en.wikipedia.org/wiki/Leap_year">Leap years</a>
```

В каждой из рассматриваемых здесь форм ссылок можно указать дополнительную метку, которая появится в виде привязки ссылки. Если опустить метку, пользователь увидит имя целевого кода или URL в качестве привязки.

Если за дескриптором `@see` следует знак **"**, то в разделе документации "См. также" отображается текст в кавычках, как показано ниже. Одно языковое средство может быть снабжено несколькими дескрипторами `@see`, но они должны быть указаны вместе.

```
@see "Core Java for the Impatient"
```

По желанию гиперссылки на другие классы или методы можно размещать в любом месте документирующих комментариев. Так, дескриптор в форме

```
{@link пакет.класс#метка_средства}
```

может быть размещен в любом месте комментария. Описание языкового средства следует тем же правилам, что и для дескриптора `@see`.

2.7.7. Комментарии к пакетам и общие комментарии

Комментарии к классам, методам и переменным размещаются непосредственно в исходных файлах Java и отделяются знаками `/** ... */`. Но для того чтобы сформировать комментарии к пакетам, в каталог каждого пакета придется ввести отдельный файл.

Этот файл называется **`package-info.java`** и должен содержать первоначальный документирующий комментарий, отделяемый знаками `/**` и `*/`, после чего следует оператор `package`. В этом файле не должно быть больше никакого кода или комментариев.

Все исходные файлы могут быть также снабжены общими комментариями. Такие комментарии размещаются в файле `overview.html` родительского каталога, содержащего все исходные файлы. Весь текст, заключенный в дескрипторы `<body>...</body>`, извлекается из общих комментариев. Эти комментарии отображаются в том случае, если пользователь выбирает кнопку Overview (Обзор) на панели навигации.

2.7.8. Извлечение комментариев

Допустим, что HTML-файлы документирующих комментариев должны быть размещены в каталоге *`docDirectory`*. Для этого нужно выполнить следующие действия.

1. Сменить каталог, содержащий исходные файлы, которые требуется задокументировать. Если требуется задокументировать вложенные пакеты, например `com.horstmann.corejava`, то следует перейти к каталогу, содержащему подкаталог `com`. (В этом каталоге должен находиться файл `overview.html`, если таковой предоставляется.)
2. Выполнить следующую команду:

   ```
   javadoc -d Каталог_документации пакет1 пакет2 ...
   ```

3. Если опустить параметр **`-d`** ***`Каталог_документации`***, HTML-файлы будут извлечены в текущий каталог. Но это может привести к путанице и поэтому не рекомендуется.

Утилита **`javadoc`** допускает дополнительную настройку с помощью многочисленных параметров командной строки. Например, чтобы включить дескрипторы в составляемую документацию, достаточно указать параметры **`-author`** и **`-version`**. (По умолчанию они опускаются.) Еще один полезный параметр **`-link`** позволяет включить гиперссылки на составляемую документацию в стандартные классы. Так, если выполнить команду

```
javadoc -link http://docs.oracle.com/javase/8/docs/api *.java
```

все классы из стандартной библиотеки будут автоматически снабжены ссылками на документацию, доступную на веб-сайте компании Oracle. Если же указать параметр **-linksource**, то каждый исходный файл будет преобразован в HTML-файл, а имя каждого класса или метода — в гиперссылку на соответствующий исходный файл.

Упражнения

1. Измените представленную в этой главе программу вывода календаря таким образом, чтобы неделя начиналась с воскресенья. Кроме того, организуйте перевод на новую строку в конце предыдущей, но только один раз.
2. Проанализируйте метод `nextInt()` из класса `Scanner`. Является ли он методом доступа или модифицирующим методом и почему? А что можно сказать о методе `nextInt()` из класса `Random`?
3. Может ли модифицирующий метод вообще возвращать что-нибудь, кроме типа `void`? Можно ли создать метод доступа с возвращаемым типом `void`, т.е. ничего фактически не возвращающий? Приведите по возможности примеры таких методов.
4. Почему в Java нельзя реализовать метод, меняющий местами содержимое двух переменных типа `int`? Вместо этого напишите метод, меняющий местами содержимое двух объектов типа `IntHolder`. (Описание этого малоизвестного класса можно найти в документации на прикладной программный интерфейс Java API.) Можно ли поменять местами содержимое двух объектов типа `Integer`?
5. Реализуйте неизменяемый класс `Point`, описывающий точку на плоскости. Предоставьте его конструктор, чтобы задать конкретную точку; конструктор без аргументов, чтобы задать точку в начале координат; а также методы `getX()`, `getY()`, `translate()` и `scale()`. В частности, метод `translate()` должен перемещать точку на определенное расстояние в направлении координат ***x*** и ***y***, а метод `scale()` — изменять масштаб по обеим координатам на заданный коэффициент. Реализуйте эти методы таким образом, чтобы они возвращали новые точки в качестве результата. Например, в следующей строке кода:

   ```
   Point p = new Point(3, 4).translate(1, 3).scale(0.5);
   ```

   в переменной `p` должна быть установлена точка с координатами **(2, 3, 5)**.
6. Повторите предыдущее упражнение, но на этот раз сделайте методы `translate()` и `scale()` модифицирующими.
7. Введите документирующие комментарии в обе версии класса `Point` из предыдущих упражнений.
8. В предыдущих упражнениях для предоставления конструкторов и методов получения из класса `Point` пришлось писать часто повторяющийся код. В большинстве ИСР имеются средства, упрощающие написание повторяющегося шаблонного кода. Имеются ли такие средства в применяемой вами ИСР?

9. Реализуйте класс `Car`, моделирующий передвижение автомобиля на бензиновом топливе по оси **x**. Предоставьте методы для передвижения автомобиля на заданное количество километров, заполнения топливного бака заданным количеством литров бензина, вычисления расстояния, пройденного от начала координат, а также уровня топлива в баке. Укажите расход топлива (в км/л) в качестве параметра конструктора данного класса. Должен ли этот класс быть неизменяемым и почему?

10. Предоставьте в классе `RandomNumbers` два статических метода типа `randomElement`, получающих произвольный элемент из обычного или списочного массива целочисленных значений. (Если обычный или списочный массив пуст, должен быть возвращен нуль.) Почему эти методы нельзя сделать методами экземпляра типа `t[]` или `ArrayList<Integer>`?

11. Перепишите класс `Cal`, чтобы использовать в нем статический импорт классов `System` и `LocalDate`.

12. Создайте исходный файл `HelloWorld.java`, где класс `HelloWorld` объявляется в пакете `ch01.sec01`. Разметите его в каком-нибудь каталоге, но только *не* в подкаталоге `ch01/sec01`. Выполните из этого каталога команду **`javac HelloWorld.java`**. Получите ли вы в итоге файл класса и где именно? Затем выполните команду **`java HelloWorld`**. Что при этом произойдет и почему? (*Подсказка:* выполните команду **`javap HelloWorld`** и проанализируйте полученное предупреждающее сообщение.) И наконец, попробуйте выполнить команду **`javac -d . HelloWorld.java`**. Почему такой способ лучше?

13. Загрузите архивный JAR-файл с библиотекой OpenCSV по адресу `http://opencsv.sourceforge.net`. Напишите класс с методом `main()` для чтения избранных файлов формата CSV и вывода некоторого их содержимого. Соответствующий образец кода можно найти на веб-сайте библиотеки OpenCSV по указанному выше адресу. А поскольку вы еще не научились обрабатывать исключения, то воспользуйтесь следующим заголовком для метода `main()`:

    ```
    public static void main(String[] args) throws Exception
    ```

    Назначение данного упражнения — не сделать что-нибудь полезное с файлами формата CSV, но поупражняться в применении библиотеки, предоставляемой в упомянутом выше архивном JAR-файле.

14. Скомпилируйте класс `Network`, представленный в этой главе. Обратите внимание на то, что файл внутреннего класса называется `Network$Member.class`. Воспользуйтесь утилитой **`javap`**, чтобы исследовать сгенерированный код. Так, по следующей команде:

    `javap -private` ***`имяКласса`***

    выводятся методы и переменные экземпляра. Выявите среди выводимых результатов ссылку на объемлющий класс. (В Linux и Mac OS X перед знаком **$** в имени класса следует указать знак **** при выполнении утилиты **`javap`**.)

15. Реализуйте полностью класс `Invoice`, представленный в разделе 2.6.1. Предоставьте метод, выводящий счет-фактуру, и демонстрационную версию программы, составляющей и выводящей образец счета-фактуры.

16. Реализуйте в классе `Queue` неограниченную очередь символьных строк. Предоставьте метод `add()` для ввода элемента в хвост очереди и метод `remove()` для удаления элемента из головы очереди. Организуйте хранение элементов в виде связного списка узлов, создав вложенный класс `Node`. Должен ли этот класс быть статическим?

17. Предоставьте *итератор* — объект, извлекающий по порядку элементы очереди из предыдущего упражнения. Сделайте класс `Iterator` вложенным и определите в нем методы `next()` и `hasNext()`. Определите в классе `Queue` метод `iterator()`, возвращающий ссылку на объект `Queue.Iterator`. Должен ли внутренний класс `Iterator` быть статическим?

Глава 3

Интерфейсы и лямбда-выражения

В этой главе...

Язык Java был разработан в 1990-е годы как объектно-ориентированный, когда принципы ООП заняли господствующее положение в разработке программного обеспечения. Интерфейсы являются ключевыми языковыми средствами ООП. Они дают возможность указать, что именно должно быть сделано, не предоставляя конкретную реализацию.

Задолго до появления ООП существовали языки функционального программирования вроде Lisp, где главную роль в структурировании кода выполняли функции, а не объекты. И только совсем недавно функциональное программирование вновь приобрело значение, поскольку оно отлично подходит для параллельного и событийно-ориентированного программирования (называемого иначе "реактивным"). В языке Java поддерживаются функциональные выражения, которые наводят удобный мост между объектно-ориентированным и функциональным программированием. В этой главе рассматриваются интерфейсы и лямбда-выражения.

Основные положения этой главы приведены ниже.

1. В интерфейсе определяется ряд методов, которые должен предоставлять класс, реализующий этот интерфейс.
2. Интерфейс является подтипом любого класса, который его реализует. Следовательно, интерфейсы отдельного класса можно присваивать переменным интерфейсного типа.
3. Интерфейс может содержать статические методы. Все переменные интерфейса автоматически являются статическими и конечными.
4. Интерфейс может содержать методы по умолчанию, которые может наследовать или переопределять реализующий его класс.
5. Интерфейсы `Comparable` и `Comparator` служат для сравнения объектов.
6. Лямбда-выражение обозначает блок кода, который может быть выполнен в последующий момент времени.
7. Лямбда-выражения преобразуются в функциональные интерфейсы.
8. По ссылкам на методы и конструкторы происходит обращение к методам и конструкторам без их вызова.
9. Лямбда-выражения и локальные внутренние классы могут эффективно получать доступ к конечным переменным из объемлющей области действия.

3.1. Интерфейсы

Интерфейс — это механизм, подробно выписывающий контракт между двумя сторонами: поставщиком службы и классами, объекты которых требуется использовать в этой службе. В последующих разделах будет показано, как определять интерфейсы и пользоваться ими в Java.

3.1.1. Объявление интерфейса

Рассмотрим следующую службу, обрабатывающую последовательности целочисленных значений, сообщая среднее первых **n** значений:

```
public static double average(IntSequence seq, int n)
```

Такие последовательности могут принимать самые разные формы. Ниже приведены некоторые их примеры.

- Последовательность целочисленных значений, предоставляемых пользователем.
- Последовательность случайных целочисленных значений.
- Последовательность простых чисел.
- Последовательность элементов целочисленного массива.
- Последовательность кодовых точек в символьной строке.
- Последовательность цифр в числе.

Для обращения со всеми этими разновидностями последовательностей требуется *единый механизм*. Прежде всего подробно разберем, что общего у целочисленных последовательностей. Для обращения с такими последовательностями требуются как минимум два метода:

- проверяющий наличие следующего элемента;
- получающий следующий элемент.

Чтобы объявить интерфейс, следует предоставить заголовки этих методов, как показано ниже.

```
public interface IntSequence {
  boolean hasNext();
  int next();
}
```

Реализовывать эти методы не нужно, но при желании можно предоставить их реализации по умолчанию, как поясняется далее, в разделе 3.2.2. Если же никакой реализации метода не предоставляется, то такой метод называется *абстрактным*.

НА ЗАМЕТКУ. Все методы из интерфейса автоматически являются открытыми (**public**). Следовательно, объявлять методы **hasNext()** и **next()** как **public** совсем не обязательно. Впрочем, некоторые программисты делают это ради большей ясности.

Методов из интерфейса достаточно, чтобы реализовать метод `average()` следующим образом:

```
public static double average(IntSequence seq, int n) {
  int count = 0;
  double sum = 0;

  while (seq.hasNext() && count < n) {
    count++;
```

```
    sum += seq.next();
  }
  return count == 0 ? 0 : sum / count;
}
```

3.1.2. Реализация интерфейса

А теперь рассмотрим другую сторону упомянутого выше контракта: классы, объекты которых требуется использовать в методе `average()`, воплощающем соответствующую службу. Эти классы должны *реализовать* интерфейс `IntSequence`. Ниже приведен пример такого класса.

```
public class SquareSequence implements IntSequence {
  private int i;

  public boolean hasNext() {
    return true;
  }

  public int next() {
    i++;
    return i * i;
  }
}
```

Существует бесконечно большое количество квадратов целых чисел, и объект данного класса предоставляет все эти квадраты по очереди. Ключевое слово `implements` обозначает, что класс `SquareSequence` намерен соответствовать интерфейсу `IntSequence`.

ВНИМАНИЕ. В классе, реализующем интерфейс, методы последнего должны быть объявлены как **`public`**. В противном случае они будут по умолчанию доступны на уровне пакета. А компилятор выдаст сообщение об ошибке, поскольку интерфейсу требуется открытый доступ.

В следующем примере кода получается среднее из 100 квадратов целых чисел:

```
SquareSequence squares = new SquareSequence();
double avg = average(squares, 100);
```

Интерфейс `IntSequence` может быть реализован во многих классах. Например, в приведенном ниже классе получается конечная последовательность цифр положительного целого числа, начиная с младшей десятичной цифры. Так, объект `new DigitSequence(1729)` предоставляет последовательность цифр **9 2 7 1**, прежде чем метод `hasNext()` возвратит логическое значение `false`.

```
public class DigitSequence implements IntSequence {
  private int number;

  public DigitSequence(int n) {
    number = n;
  }
```

```
  public boolean hasNext() {
    return number != 0;
  }

  public int next() {
    int result = number % 10;
    number /= 10;
    return result;
  }

  public int rest() {
    return number;
  }
}
```

НА ЗАМЕТКУ. Классы **SquareSequence** и **DigitSequence** реализуют все методы из интерфейса **IntSequence**. Если же класс реализует только некоторые из методов интерфейса, он должен быть объявлен с модификатором доступа **abstract**. Подробнее об абстрактных классах речь пойдет в главе 4.

3.1.3. Преобразование в интерфейсный тип

В следующем фрагменте кода вычисляется среднее последовательности цифр:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
  // Обработаны будут только первые четыре значения из последовательности
```

Обратите внимание на переменную `digits`. Она относится к типу `IntSequence`, а не `DigitSequence`. Переменная типа `IntSequence` ссылается на объект некоторого класса, реализующего интерфейс `IntSequence`. Объект всегда можно присвоить переменной, тип которой относится к реализованному интерфейсу, или передать его методу, ожидающему такой интерфейс в качестве параметра.

Обратимся к более удобной терминологии. Тип **S** служит *супертипом* для типа **T** (называемого в этом случае *подтипом*), если любое значение подтипа может быть присвоено переменной супертипа без преобразования. Например, интерфейс `IntSequence` служит супертипом для класса `DigitSequence`.

НА ЗАМЕТКУ. Несмотря на то что объявлять переменные интерфейсного типа вполне допустимо, объект такого типа получить вообще нельзя. Все объекты являются экземплярами классов.

3.1.4. Приведение типов и операция `instanceof`

Иногда требуется противоположное преобразование из супертипа в подтип. Для этой цели служит *приведение типов*. Так, если заранее известно, что объект, хранящийся в переменной типа `IntSequence`, на самом деле относится к типу `DigitSequence`, то его тип можно преобразовать следующим образом:

```
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

В этом случае требуется приведение типов, поскольку метод `rest()` относится к классу `DigitSequence`, а не к интерфейсу `IntSequence`. Более убедительный пример такого преобразования типов приведен в упражнении 2 в конце этой главы.

Объект можно привести только к типу его конкретного класса или его подклассов. В противном случае возникнет ошибка во время компиляции или исключение в связи с неверным приведением к типу класса во время выполнения.

```
String digitString = (String) sequence;
  // Может и не сработать, поскольку интерфейс IntSequence
  // не служит супертипом для класса String
RandomSequence randoms = (RandomSequence) sequence;
  // Может сработать, а иначе возникнет исключение
  // в связи с неверным приведением к типу класса
```

Во избежание исключения необходимо сначала проверить, относится *ли* объект к нужному типу, используя операцию **`instanceof`**. Так, в результате вычисления выражения

`объект` **`instanceof`** ***`Тип`***

возвращается логическое значение `true`, если заданный ***`объект`*** является экземпляром класса, у которого указанный ***`Тип`*** имеется в качестве супертипа. Такую проверку целесообразно выполнять перед приведением типов, как показано ниже.

```
if (sequence instanceof DigitSequence) {
  DigitSequence digits = (DigitSequence) sequence;
  ...
}
```

3.1.5. Расширение интерфейсов

Один интерфейс может *расширять* другой интерфейс, чтобы предоставить дополнительные методы, помимо уже имеющихся в последнем. В качестве примера ниже приведен интерфейс `Closeable` с единственным методом.

```
public interface Closeable {
  void close();
}
```

Как будет показано в главе 5, этот интерфейс очень важен для закрытия ресурсов при возникновении исключения. Этот интерфейс расширяется интерфейсом `Channel` следующим образом:

```
public interface Channel extends Closeable {
  boolean isOpen();
}
```

Класс, реализующий интерфейс Channel, должен предоставить оба метода. А его объекты могут быть преобразованы к типам обоих интерфейсов — Closeable и Channel.

3.1.6. Реализация нескольких интерфейсов

Класс может реализовать любое количество интерфейсов. Например, класс FileSequence, предназначенный для чтения целых чисел из файла, может реализовать интерфейс Closeable, помимо интерфейса IntSequence, как показано ниже. В таком случае у класса FileSequence имеются оба интерфейса, IntSequence и Closeable, как его супертипы.

```
public class FileSequence implements IntSequence, Closeable {
  ...
}
```

3.17. Константы

Любая переменная, определенная в интерфейсе, автоматически объявляется как public static final, т.е. как константа. Так, в следующем примере интерфейса SwingConstants определяются константы для обозначения направлений по компасу:

```
public interface SwingConstants {
  int NORTH = 1;
  int NORTH_EAST = 2;
  int EAST = 3;
  ...
}
```

Обращаться к этим константам можно по их полностью уточненному имени, например SwingConstants.NORTH. Если в классе решено реализовать интерфейс SwingConstants, то описатель SwingConstants можно опустить и просто написать имя константы NORTH. Но такой подход мало распространен. Ведь для установки констант намного удобнее пользоваться перечислениями, как поясняется в главе 4.

НА ЗАМЕТКУ. Наличие переменных экземпляра в интерфейсе не предусмотрено. Ведь интерфейс определяет поведение, а не состояние объекта.

3.2. Методы статические и по умолчанию

В первых версиях Java все методы в интерфейсе должны были быть абстрактными, т.е. они не имели тела. А ныне в интерфейсе допускаются следующие две разновидности методов с конкретной реализацией: статические и по умолчанию. Обе эти разновидности интерфейсных методов рассматриваются по очереди в последующих разделах.

3.2.1. Статические методы

Формальных причин, по которым в интерфейсе не могли бы присутствовать статические методы, никогда не существовало. Но такие методы не согласовывались с представлением об интерфейсах как об абстрактных спецификациях. Это представление теперь изменилось. В частности, имеется немало причин определять фабричные методы в интерфейсах. Так, в интерфейсе `IntSequence` может быть объявлен статический метод `digitsOf()`, формирующий последовательность цифр заданного числа следующим образом:

```
IntSequence digits = IntSequence.digitsOf(1729);
```

Этот метод возвращает экземпляр некоторого класса, реализующего интерфейс `IntSequence`, как показано ниже. Но в вызывающем коде совсем не обязательно знать, о каком классе идет речь.

```
public interface IntSequence {
  ...
  public static IntSequence digitsOf(int n) {
    return new DigitSequence(n);
  }
}
```

НА ЗАМЕТКУ. В прошлом статические методы зачастую определялись в дополнительном классе, сопутствующем интерфейсу. В стандартной библиотеке Java можно обнаружить пары интерфейсов и служебных классов, например **`Collection`/`Collections`** или **`Path`/`Paths`**. Такое разделение больше не требуется.

3.2.2. Методы по умолчанию

Для любого интерфейсного метода можно предоставить реализацию *по умолчанию*. Такой метод следует пометить модификатором доступа `default`, как показано ниже. В классе, реализующем приведенный ниже интерфейс, метод `hasNext()` может быть переопределен, или же унаследована его реализация по умолчанию.

```
public interface IntSequence {
  default boolean hasNext() { return true; }
    // По умолчанию последовательности бесконечны
  int next();
}
```

НА ЗАМЕТКУ. Методы по умолчанию положили конец предоставлению по классическому шаблону интерфейса и сопутствующего ему класса, реализующего большую часть или все его методы, например, пары интерфейсов и служебных классов **`Collection`/`AbstractCollection`** или **`WindowListener`/`WindowAdapter`** в прикладном программном интерфейсе Java API. Теперь достаточно реализовать методы в самом интерфейсе.

Методы по умолчанию играют важную роль в дальнейшем *развитии интерфейсов*. Рассмотрим в качестве примера интерфейс `Collection`, многие годы входящий в состав стандартной библиотеки Java. Допустим, что некогда был предоставлен следующий класс, реализующий интерфейс `Collection`:

```
public class Bag implements Collection
```

а впоследствии, начиная с версии Java 8, в этот интерфейс был внедрен метод `stream()`.

Допустим также, что метод `stream()` не является методом по умолчанию. В таком случае класс `Bag` больше не компилируется, поскольку он не реализует новый метод из интерфейса `Collection`. Таким образом, внедрение в интерфейс метода не по умолчанию нарушает *совместимость на уровне исходного кода*.

Но допустим, что этот класс не перекомпилируется и просто используется содержащий его старый архивный JAR-файл. Этот класс по-прежнему загружается, несмотря на отсутствующий в нем метод. В программах могут по-прежнему строиться экземпляры класса `Bag`, и ничего плохого не произойдет. (Внедрение метода в интерфейс *совместимо на уровне двоичного кода*.) Но если в программе делается вызов метода `stream()` для экземпляра класса `Bag`, то возникает ошибка типа `AbstractMethodError`.

Эти затруднения можно устранить, если объявить метод `stream()` как `default`. И тогда класс `Bag` будет компилироваться снова. А если этот класс загружается без перекомпиляции и метод `stream()` вызывается для экземпляра класса `Bag`, то такой вызов происходит по ссылке `Collection.stream`.

3.2.3. Разрешение конфликтов с методами по умолчанию

Если класс реализует два интерфейса, в одном из которых имеется метод по умолчанию, а в другом метод (по умолчанию или иной) с таким же самым именем и типами параметров, то подобный конфликт должен быть непременно разрешен. Он возникает нечасто и обычно разрешается легко.

Обратимся к конкретному примеру. Допустим, имеется следующий интерфейс `Person` с методом по умолчанию `getId()`:

```
public interface Person {
  String getName();
  default int getId() { return 0; }
}
```

Допустим также, что имеется следующий интерфейс `Identified` с таким же методом по умолчанию:

```
public interface Identified {
  default int getId() { return Math.abs(hashCode()); }
}
```

О назначении метода `hashCode()` речь пойдет в главе 4, а до тех пор достаточно сказать, что он возвращает некоторое целочисленное значение, производное от объекта. Что произойдет, если сформировать приведенный ниже класс, реализующий оба интерфейса?

```
public class Employee implements Person, Identified {
  ...
}
```

Этот класс наследует оба метода `getId()`, предоставляемых интерфейсами `Person` и `Identified`. Но у компилятора Java нет критерия предпочесть один из этих методов другому, и поэтому он выдаст сообщение об ошибке, предоставив нам самим разрешать возникшую неоднозначность. С этой целью можно предоставить метод `getId()` в классе `Employee` и реализовать собственную схему идентификаторов или поручить это одному из конфликтующих методов следующим образом:

```
public class Employee implements Person, Identified {
  public int getId() { return Identified.super.getId(); }
  ...
}
```

НА ЗАМЕТКУ. Ключевое слово **`super`** позволяет вызывать метод супертипа. В данном случае требуется указать нужный супертип. Подобный синтаксис может показаться не совсем обычным, но он согласуется с синтаксисом для вызова метода из суперкласса, как поясняется в главе 4.

А теперь допустим, что интерфейс `Identified` не предоставляет реализацию метода `getId()` по умолчанию, как показано ниже.

```
interface Identified {
  int getId();
}
```

Может ли класс `Employee` унаследовать метод по умолчанию из интерфейса `Person`? На первый взгляд это может показаться вполне обоснованным. Но откуда компилятору знать, что метод `Person.getId()` на самом деле делает то, что ожидается от метода `Identified.getId()`? Ведь этот метод может возвратить уровень подсознания личности по Фрейду, а не идентификационный номер работника.

Разработчики Java решили сделать выбор в пользу безопасности и единообразия. Не имеет значения, каким образом конфликтуют оба интерфейса. Если хотя бы один из интерфейсов предоставляет реализацию метода, то компилятор сообщает об ошибке, а неоднозначность должен разрешить сам программист.

НА ЗАМЕТКУ. Если ни один из интерфейсов не предоставляет реализацию по умолчанию общего для них метода, то никакого конфликта не возникает. В таком случае для класса, реализующего эти интерфейсы, имеются два варианта выбора: реализовать метод или оставить его нереализованным и объявить класс как **`abstract`**.

НА ЗАМЕТКУ. Если класс расширяет суперкласс (см. главу 4) и реализует интерфейс, наследуя один и тот же метод из интерфейса и суперкласса, то правила разрешения конфликтов упрощаются. В этом случае имеет значение метод из суперкласса, а любой метод по умолчанию из интерфейса просто игнорируется. На самом деле это более распространенный случай, чем конфликт интерфейсов. Подробнее об этом — в главе 4.

3.3. Примеры интерфейсов

На первый взгляд кажется, что интерфейсы способны не на многое. Ведь в интерфейсе лишь определяется ряд методов, которые класс обязуется реализовать. Чтобы важность интерфейсов стала более очевидной, в последующих разделах демонстрируются четыре примера часто употребляемых интерфейсов из стандартной библиотеки Java.

3.3.1. Интерфейс Comparable

Допустим, что требуется отсортировать массив объектов. Алгоритм сортировки предполагает неоднократно повторяющееся сравнение элементов массива и их переупорядочение, если они располагаются не по порядку. Разумеется, правила сравнения для каждого класса разные, и алгоритм сортировки должен предусматривать лишь вызов метода, предоставляемого классом. Если во всех классах может быть согласован вызов конкретного метода, то алгоритм сортировки справляется со своей задачей. Именно здесь и приходит на помощь интерфейс.

Если в классе требуется разрешить сортировку его объектов, он должен реализовать интерфейс `Comparable`. У этого интерфейса имеется своя особенность. Обычно требуется сравнивать разные объекты: символьные строки, работников и пр. Именно поэтому у интерфейса `Comparable` имеется параметр типа:

```
public interface Comparable<T> {
  int compareTo(T other);
}
```

Например, класс `String` реализует интерфейс `Comparable<String>`, чтобы у его метода `compareTo()` была следующая сигнатура:

```
int compareTo(String other)
```

НА ЗАМЕТКУ. Параметр типа, например, в интерфейсе **`Comparable`** или **`ArrayList`**, имеет тип, называемый *обобщенным*. Подробнее об обобщенных типах речь пойдет в главе 6.

В результате вызова `x.compareTo(y)` метод `compareTo()` возвращает целочисленное значение, обозначающее порядок следования сравниваемых объектов `x` и `y`. Так, если возвращается положительное значение (не обязательно **`1`**), то объект `x` должен следовать после объекта `y`. А если возвращается отрицательное значение (не обязательно **`-1`**), то объект `x` должен следовать перед объектом `y`. Если же объекты `x` и `y` считаются одинаковыми, то возвращается нуль.

Следует иметь в виду, что возвращаемое значение может быть любым целым числом. Это очень удобно, поскольку дает возможность возвращать разность неотрицательных целочисленных значений, как показано ниже.

```
public class Employee implements Comparable<Employee> {
  ...
  public int compareTo(Employee other) {
```

```
    return getId() - other.getId(); // Верно, если идентификаторы всегда ≥ 0
  }
}
```

ВНИМАНИЕ. Возвратить разность неотрицательных целочисленных значений не удастся, если эти значения отрицательные. Ведь если вычитать крупные операнды с противоположным знаком, то может возникнуть переполнение. В таком случае следует воспользоваться методом **`Integer.compare()`**, правильно обращающимся со всеми целочисленными значениями.

При сравнении числовых значений с плавающей точкой нельзя просто возвратить их разность. Вместо этого следует воспользоваться методом `Double.compare()`, правильно обращающимся со сравниваемыми значениями, даже если это $\pm\infty$ и не число (NaN).

В следующем примере кода показано, как реализовать в классе `Employee` интерфейс `Comparable`, чтобы упорядочить работников по размеру зарплаты:

```
public class Employee implements Comparable<Employee> {
  ...
  public int compareTo(Employee other) {
    return Double.compare(salary, other.salary);
  }
}
```

НА ЗАМЕТКУ. Доступ к переменной экземпляра по ссылке **`other.salary`** в методе **`compare()`** вполне допустим. В языке Java методу доступны закрытые элементы *любого* объекта его класса.

Класс `String`, как и многие другие классы из стандартной библиотеки Java, реализует интерфейс `Comparable`. Так, для сортировки массива объектов типа `String` можно воспользоваться методом `Arrays.sort()` следующим образом:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends); // массив friends теперь упорядочен
                      // следующим образом: ["Mary", "Paul", "Peter"]
```

НА ЗАМЕТКУ. Как ни странно, метод **`Arrays.sort()`** не проверяет во время компиляции, относится ли его аргумент к массиву объектов типа **`String`**. Вместо этого генерируется исключение, если в этом методе встретится элемент класса, не реализующего интерфейс **`Comparable`**.

3.3.2. Интерфейс `Comparator`

А теперь допустим, что требуется отсортировать символьные строки по порядку увеличения их *длины*, а не в лексикографическом порядке. В классе `String` нельзя реализовать метод `compareTo()` двумя разными способами. К тому же этот класс относится к стандартной библиотеке Java и не подлежит изменению. В качестве выхода из этого положения можно воспользоваться вторым вариантом метода `Arrays.sort()`, параметрами которого являются массив и *компаратор* — экземпляр класса, реализующего приведенный ниже интерфейс `Comparator`.

```
public interface Comparator<T> {
  int compare(T first, T second);
}
```

Чтобы сравнить символьные строки по длине, достаточно определить класс, реализующий интерфейс `Comparator<String>`:

```
class LengthComparator implements Comparator<String> {
  public int compare(String first, String second) {
    return first.length() - second.length();
  }
}
```

Чтобы провести сравнение, фактически требуется получить экземпляр данного класса следующим образом:

```
Comparator<String> comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) ...
```

Сравните этот фрагмент кода с вызовом `words[i].compareTo(words[j])`. Метод `compare()` вызывается для объекта компаратора, а не для самой символьной строки.

НА ЗАМЕТКУ. Несмотря на то что у объекта типа **`LengthComparator`** отсутствует состояние, его экземпляр все же требуется получить, чтобы вызвать метод **`compare()`**. Ведь этот метод не является статическим.

Чтобы отсортировать массив, достаточно передать объект типа `LengthComparator` методу `Arrays.sort()` следующим образом:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

Теперь массив упорядочен как `["Paul", "Mary", "Peter"]` или `["Mary", "Paul", "Peter"]`. Далее, в разделе 3.4.2, будет показано, как лямбда-выражения упрощают пользование интерфейсом `Comparator`.

3.3.3. Интерфейс `Runnable`

Теперь, когда практически все процессоры состоят как минимум из двух ядер, последние нужно каким-то образом задействовать, чтобы выполнять некоторые задачи в отдельных потоках или предоставить им пул потоков для параллельного выполнения. Для решения подобных задач следует реализовать интерфейс `Runnable`, как показано ниже. У этого интерфейса имеется единственный метод `run()`.

```
class HelloTask implements Runnable {
  public void run() {
    for (int i = 0; i < 1000; i++) {
      System.out.println("Hello, World!");
    }
  }
}
```

Если задачу требуется выполнить в новом потоке, достаточно создать поток исполнения из интерфейса `Runnable` и запустить его, как показано ниже. Теперь метод `run()` выполняется в отдельном потоке, тогда как в текущем потоке может быть продолжено исполнение другого задания.

```
Runnable task = new HelloTask();
Thread thread = new Thread(task);
thread.start();
```

НА ЗАМЕТКУ. Другие способы исполнения потоков из интерфейса **`Runnable`** представлены в главе 10.

НА ЗАМЕТКУ. Для выполнения задач, возвращающих результат типа **`T`**, имеется также интерфейс **`Callable<T>`**.

3.3.4. Обратные вызовы в пользовательском интерфейсе

В графическом пользовательском интерфейсе (ГПИ) можно указать действия, которые должны выполняться, когда пользователь щелкает на экранной кнопке, выбирает пункт меню, перемещает ползунок и т.д. Такие действия обычно называют *обратными вызовами*, поскольку они приводят к обратному вызову некоторого кода, когда происходит какое-нибудь действие пользователя.

В библиотеках Java, предназначенных для построения ГПИ, интерфейсы служат для обратных вызовов. Например, следующий интерфейс из библиотеки JavaFX служит для извещения о событиях, наступающих в ГПИ:

```
public interface EventHandler<T> {
  void handle(T event);
}
```

Это слишком обобщенный интерфейс, где **`T`** обозначает тип извещаемого события, например, `ActionEvent` для щелчка на экранной кнопке. Чтобы указать действие, нужно сначала реализовать интерфейс следующим образом:

```
class CancelAction implements EventHandler<ActionEvent> {
  public void handle(ActionEvent event) {
    System.out.println("Oh noes!");
  }
}
```

а затем создать объект данного класса и ввести его в код обработки событий от экранной кнопки таким образом:

```
Button cancelButton = new Button("Cancel");
cancelButton.setOnAction(new CancelAction());
```

НА ЗАМЕТКУ. Компания Oracle позиционирует библиотеку JavaFX как преемницу библиотеки Swing для построения ГПИ, и поэтому в представленных здесь примерах применяется библиотека JavaFX. Подробности реализации особого значения не имеют, поскольку экранная кнопка снабжается

некоторым кодом, который должен выполняться при ее нажатии, независимо от конкретной библиотеки, применяемой для построения ГПИ, будь то JavaFX, Swing или AWT.

Безусловно, такой способ определения действия экранной кнопки довольно трудоемок. В других языках программирования для экранной кнопки достаточно вызвать соответствующую функцию, не создавая класс и не получая его экземпляр. В следующем разделе поясняется, как то же самое можно сделать и в Java.

3.4. Лямбда-выражения

Лямбда-выражение — это блок кода, который передается для последующего выполнения один или несколько раз. В предыдущих разделах рассматривались случаи, когда такой блок кода оказался бы очень кстати, в том числе для:

- передачи метода сравнения методу `Arrays.sort()`;
- выполнения задачи в отдельном потоке;
- указания действия, которое должно быть выполнено после щелчка на экранной кнопке.

Но Java — объектно-ориентированный язык программирования, где почти все является объектом. В языке Java отсутствуют функциональные типы данных. Вместо этого функции выражаются в виде объектов, экземпляров классов, реализующих конкретный интерфейс. Лямбда-выражения предоставляют удобный синтаксис для получения таких экземпляров.

3.4.1. Синтаксис лямбда-выражений

Обратимся снова к примеру кода из раздела 3.3.2. В этом коде определяется, является ли одна символьная строка короче другой. С этой целью вычисляется следующее выражение:

```
first.length() - second.length()
```

А что обозначают ссылки `first` и `second`? Они обозначают символьные строки. Язык Java является строго типизированным, и поэтому приведенное выше выражение можно написать следующим образом:

```
(String first, String second) -> first.length() - second.length()
```

Собственно говоря, это и есть *лямбда-выражение*. Такое выражение представляет собой блок кода вместе с указанием любых переменных, которые должны быть переданы коду.

А почему оно так называется? Много лет назад, задолго до появления компьютеров, математик и логик Алонсо Черч (Alonzo Church) формализовал, что же должно означать эффективное выполнение математической функции. (Любопытно, что имеются такие функции, о существовании которых известно, но никто не знает, как вычислить их значения.) Он употребил греческую букву лямбда (λ) для обозначения параметров функции аналогично следующему:

```
lfirst. lsecond. first.length() - second.length()
```

НА ЗАМЕТКУ. А почему была выбрана именно буква λ? Неужели Алонсо Черчу не хватило букв латинского алфавита? По давней традиции знаком ударения (^) в математике обозначаются параметры функции, что навело Алонсо Черча на мысль воспользоваться прописной буквой Λ. Но в конечном итоге он остановил свой выбор на строчной букве λ. И с тех пор выражение с переменными параметрами называют лямбда-выражением.

Если в теле лямбда-выражения должно быть выполнено вычисление, которое не вписывается в одно выражение, его можно написать точно так же, как и тело метода, заключив в фигурные скобки {} и явно указав операторы `return`, как в следующем примере кода:

```
(String first, String second) -> {
  int difference = first.length() < second.length();
  if (difference < 0) return -1;
  else if (difference > 0) return 1;
  else return 0;
}
```

Если у лямбда-выражения отсутствуют параметры, следует указать пустые круглые скобки, как при объявлении метода без параметров:

```
Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }
```

Если же типы параметров лямбда-выражения могут быть выведены, их можно опустить, как в следующем примере кода:

```
Comparator<String> comp =
    (first, second) -> first.length() - second.length();
      // То же, что и (String first, String second)
```

В данном примере компилятор может сделать вывод, что ссылки `first` и `second` должны обозначать символьные строки, поскольку результат вычисления лямбда-выражения присваивается компаратору символьных строк. (Эту операцию присваивания мы рассмотрим более подробно в следующем разделе.)

Если у метода имеется единственный параметр выводимого типа, то можно даже опустить круглые скобки, как показано ниже.

```
EventHandler<ActionEvent> listener = event ->
  System.out.println("Oh noes!");
    // Вместо выражения (event) -> или (ActionEvent event) ->
```

Результат вычисления лямбда-выражения вообще не указывается. Тем не менее компилятор выводит его из тела лямбда-выражения и проверяет, соответствует ли он ожидаемому результату. Например, выражение

```
(String first, String second) -> first.length() - second.length()
```

может быть использовано в контексте, где ожидается результат типа `int` (или совместимого с ним типа вроде `Integer`, `long`, `double`).

3.4.2. Функциональные интерфейсы

Как было показано ранее, в Java имеется немало интерфейсов, выражающих действия, в том числе `Runnable` и `Comparator`. Лямбда-выражения совместимы с этими интерфейсами. Лямбда-выражение можно предоставить всякий раз, когда ожидается объект класса, реализующего интерфейс с *единственным абстрактным методом*. Такой интерфейс называется *функциональным*.

Чтобы продемонстрировать преобразование в функциональный интерфейс, рассмотрим снова метод `Arrays.sort()`. В качестве второго параметра ему требуется экземпляр типа `Comparator` — интерфейса с единственным методом. Вместо него достаточно предоставить лямбда-выражение следующим образом:

```
Arrays.sort(words,
    (first, second) -> first.length() - second.length());
```

Переменная второго параметра метода `Arrays.sort()` автоматически принимает объект некоторого класса, реализующего интерфейс `Comparator<String>`. Управление такими объектами и классами полностью зависит от конкретной реализации и оптимизировано в достаточной степени.

В большинстве языков программирования, поддерживающих функциональные литералы, можно объявлять типы функций вроде `(String, String) -> int`, объявлять переменные подобных типов, присваивать этим переменным функции и вызывать их. А с лямбда-выражением в Java можно делать *лишь одно*: присваивать его переменной, типом которой является функциональный интерфейс, чтобы преобразовать его в экземпляр данного интерфейса.

НА ЗАМЕТКУ. Лямбда-выражение нельзя присвоить переменной типа `Object`, т.е. общего супертипа для всех классов в Java (см. главу 4). `Object` — это класс, а не функциональный интерфейс.

В стандартной библиотеке Java предоставляется немало функциональных интерфейсов (см. раздел 3.6.2). Ниже приведен один из них.

```
public interface Predicate<T> {
  boolean test(T t);
  // Дополнительные методы по умолчанию и статические методы
}
```

В классе `ArrayList` имеется метод `removeIf()` с параметром типа `Predicate`. Этот метод специально предназначен для передачи ему лямбда-выражения. Например, в следующем выражении из списочного массива удаляются все пустые (`null`) значения:

```
list.removeIf(e -> e == null);
```

3.5. Ссылки на методы и конструкторы

Иногда уже имеется метод, выполняющий именно то действие, которое требуется передать какому-нибудь другому коду. Для ссылки на метод, которая оказывается еще короче, чем лямбда-выражение, вызывающее метод, предусмотрен специальный

синтаксис. Аналогичная ссылка существует и на конструктор. Обе разновидности этих ссылок рассматриваются по очереди в последующих разделах.

3.5.1. Ссылки на методы

Допустим, что символьные строки требуется отсортировать независимо от регистра букв. С этой целью можно было бы сделать следующий вызов:

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));
```

Вместо этого данное выражение-метод можно передать следующим образом:

```
Arrays.sort(strings, String::compareToIgnoreCase);
```

Выражение `String::compareToIgnoreCase` обозначает *ссылку на метод*, которая равнозначна лямбда-выражению `(x, y) -> x.compareToIgnoreCase(y)`.

Рассмотрим еще один пример. В классе `Objects` определяется метод `isNull()`. Так, в результате вызова `Objects.isNull(x)` возвращается значение `x == null`. На первый взгляд для подобной операции вряд ли требуется отдельный метод, но он специально предназначен для передачи ему ссылки на метод. Например, в результате следующего вызова:

```
list.removeIf(Objects::isNull);
```

из списка удаляются все пустые значения.

В качестве еще одного примера допустим, что требуется вывести все элементы списка. В классе `ArrayList` имеется метод `forEach()`, применяющий передаваемую ему функцию к каждому элементу списка. Его можно вызвать, например, следующим образом:

```
list.forEach(x -> System.out.println(x));
```

Но было бы намного изящнее передать метод `println()` методу `forEach()`. Это можно сделать следующим образом:

```
list.forEach(System.out::println);
```

Как следует из приведенных выше примеров, операция `::` отделяет имя метода от имени класса или объекта. Ниже приведены три разновидности этой операции.

- ***Класс*::*МетодЭкземпляра***
- ***Класс*::*СтатическийМетод***
- ***Объект*::*МетодЭкземпляра***

В первом случае первый параметр становится получателем метода, а все остальные параметры передаются методу. Например, ссылка на метод `String::compareToIgnoreCase` равнозначна лямбда-выражению `(x, y) -> x.compareToIgnoreCase(y)`.

Во втором случае все параметры передаются статическому методу. Так, ссылка на метод `Objects::isNull` равнозначна лямбда-выражению `x -> Objects.isNull(x)`.

В третьем случае метод вызывается для заданного объекта, а параметры передаются методу экземпляра. Следовательно, ссылка на метод `System.out::println` равнозначна лямбда-выражению `x -> System.out.println(x)`.

НА ЗАМЕТКУ. Если имеется несколько переопределяемых методов с одинаковым именем, то компилятор попытается выяснить из контекста назначение каждого из них. Например, имеется несколько вариантов метода **`println()`**. Когда же ссылка на этот метод передается методу **`forEach()`** из класса **`ArrayList<String>`**, то выбирается вариант **`println(String)`**.

В ссылке на метод допускается указывать ссылку `this`. Например, ссылка на метод `this::equals` равнозначна лямбда-выражению `x -> this.equals(x)`.

НА ЗАМЕТКУ. Во внутреннем классе можно указать ссылку **`this`** на объемлющий его класс следующим образом: ***`ОбъемлющийКласс`*`::this::`*`Метод`**. Аналогично можно указать и ссылку **`super`** (см. главу 4).

3.5.2. Ссылки на конструкторы

Ссылки на конструкторы действуют таким же образом, как и ссылки на методы, за исключением того, что вместо имени метода указывается оператор `new`. Например, ссылка `Employee::new` делается на конструктор класса `Employee`. Если же у класса имеется несколько конструкторов, то конкретный конструктор выбирается по ссылке в зависимости от контекста.

Рассмотрим пример, демонстрирующий применение ссылки на конструктор. Допустим, что имеется следующий список символьных строк:

```
List<String> names = ...;
```

Требуется составить список работников по их именам. Как будет показано в главе 8, не прибегая к циклу, это можно сделать с помощью потоков данных следующим образом: сначала преобразовать список в поток данных, а затем вызвать метод `map()`. Этот метод применяет передаваемую ему функцию и накапливает полученные результаты, как показано ниже. Поток данных `names.stream()` содержит объекты типа `String`, и поэтому компилятору известно, что ссылка `Employee::new` делается на конструктор класса `Employee(String)`.

```
Stream<Employee> stream = names.stream().map(Employee::new);
```

Ссылки на массивы можно сформировать с помощью типов массивов. Например, `int[]::new` — это ссылка на конструктор с одним параметром, обозначающим длину массива. Она равнозначна лямбда-выражению `n -> new int[n]`.

Ссылки на конструкторы в виде массивов удобны для преодоления следующего ограничения: в Java нельзя построить массив обобщенного типа. (Подробнее об этом — в главе 6.) Именно по этой причине такие методы, как `Stream.toArray()`, возвращают массив типа `Object`, а не массив типа его элементов, как показано ниже.

```
Object[] employees = stream.toArray();
```

Но этого явно недостаточно. Пользователю требуется массив работников, а не объектов. В качестве выхода из этого положения можно вызвать другой вариант метода `toArray()`, принимающий ссылку на конструктор следующим образом:

```
Employee[] buttons = stream.toArray(Employee[]::new);
```

Метод `toArray()` сначала вызывает этот конструктор, чтобы получить массив правильного типа. А затем он заполняет массив и возвращает его.

3.6. Обработка лямбда-выражений

До сих пор пояснялось, как составлять лямбда-выражения и передавать их методу, ожидающему функциональный интерфейс. А в последующих разделах будет показано, как писать собственные методы, способные употреблять лямбда-выражения.

3.6.1. Реализация отложенного выполнения

Лямбда-выражения применяются для *отложенного выполнения*. Ведь если некоторый код требуется выполнить сразу, это можно сделать, не заключая его в лямбда-выражение. Для отложенного выполнения кода имеется немало причин, в том числе следующие.

- Выполнение кода в отдельном потоке.
- Неоднократное выполнение кода.
- Выполнение кода в нужный момент по ходу алгоритма (например, выполнение операции сравнения при сортировке).
- Выполнение кода при наступлении какого-нибудь события (щелчка на экранной кнопке, поступления данных и т.д.).
- Выполнение кода только по мере надобности.

Рассмотрим простой пример. Допустим, что некоторое действие требуется повторить `n` раз. Это действие и количество его повторений передаются методу `repeat()` следующим образом:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

Чтобы принять лямбда-выражение в качестве параметра, нужно выбрать (а в редких случаях — предоставить) функциональный интерфейс. В данном примере для этого достаточно воспользоваться интерфейсом `Runnable`, как показано ниже. Обратите внимание на то, что тело лямбда-выражения выполняется при вызове `action.run()`.

```
public static void repeat(int n, Runnable action) {
  for (int i = 0; i < n; i++) action.run();
}
```

А теперь немного усложним рассматриваемый здесь простой пример, чтобы уведомить действие, на каком именно шаге цикла оно должно произойти. Для этого

нужно выбрать функциональный интерфейс с методом, принимающим параметр типа int и ничего не возвращающим (void). Вместо написания собственного функционального интерфейса лучше воспользоваться одним из стандартных интерфейсов, описываемых в следующем разделе. Ниже приведен стандартный интерфейс для обработки значений типа int.

```
public interface IntConsumer {
  void accept(int value);
}
```

Усовершенствованный вариант метода repeat() выглядит следующим образом:

```
public static void repeat(int n, IntConsumer action) {
  for (int i = 0; i < n; i++) action.accept(i);
}
```

А вызывается он таким образом:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

3.6.2. Вызов функционального интерфейса

В большинстве языков функционального программирования типы функций являются *структурными*. Так, для указания функции, преобразующей две символьные строки в целое число, служит тип, аналогичный одному из следующих: Function2<String, String, Integer> или (String, String) -> int. А в языке Java вместо этого цель функции объявляется с помощью функционального интерфейса вроде Comparator<String>. В теории языков программирования это называется *номинальной типизацией*.

Безусловно, имеется немало случаев, когда требуется принять любую функцию без конкретной семантики. И для этой цели имеется целый ряд обобщенных функциональных типов (табл. 3.1). Пользоваться ими рекомендуется при всякой возможности.

Таблица 3.1. Наиболее употребительные функциональные интерфейсы

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода	Описание	Другие методы
`Runnable`	Отсутствуют	`void`	`run`	Выполняет действие без аргументов или возвращаемого значения	
`Supplier<T>`	Отсутствуют	`T`	`get`	Предоставляет значение типа `T`	
`Consumer<T>`	`T`	`void`	`accept`	Употребляет значение типа `T`	`andThen`
`BiConsumer<T, U>`	`T`, `U`	`void`	`accept`	Употребляет значения типа `T` и `U`	`andThen`

Окончание табл. 3.1

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода	Описание	Другие методы
`Function<T, R>`	`T`	`R`	`apply`	Функция с аргументом `T`	`compose`, `andThen`, `identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	Функция с аргументами `T` и `U`	`andThen`
`UnaryOperator<T>`	`T`	`T`	`apply`	Унарная операция над типом `T`	`compose`, `andThen`, `identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	Двоичная операция над типом `T`	`andThen`, `maxBy`, `minBy`
`Predicate<T>`	`T`	`boolean`	`test`	Булевозначная функция	`and`, `or`, `negate`, `isEqual`
`BiPredicate<T, U>`	`T, U`	`boolean`	`test`	Булевозначная функция с аргументами	`and`, `or`, `negate`

Допустим, что требуется написать метод для обработки файлов, соответствующих определенному критерию. Следует ли для этого воспользоваться описательным классом `java.io.FileFilter` или функциональным интерфейсом `Predicate<File>`? В этом случае настоятельно рекомендуется воспользоваться функциональным интерфейсом `Predicate<File>`. Единственная причина не делать этого — наличие многих полезных методов, получающих экземпляры типа `FileFilter`.

НА ЗАМЕТКУ. У большинства стандартных функциональных интерфейсов имеются неабстрактные методы получения или объединения функций. Например, вызов метода **`Predicate.isEqual(a)`** равнозначен ссылке на метод **`a::equals`**, но он действует и в том случае, если аргумент **`a`** имеет пустое значение **`null`**. Для объединения предикатов имеются методы по умолчанию **`and()`**, **`or()`**, **`negate()`**. Например, вызов метода **`Predicate.isEqual(a).or(Predicate.isEqual(b))`** равнозначен выражению **`x -> a.equals(x) || b.equals(x)`**.

В табл. 3.2 перечислены 34 доступные специализации примитивных типов `int`, `long` и `double`. Эти специализации рекомендуется употреблять для сокращения автоупаковки. Именно по этой причине в примере кода из предыдущего раздела использовался интерфейс `IntConsumer` вместо интерфейса `Consumer<Integer>`.

Таблица 3.2. Функциональные интерфейсы для примитивных типов, где обозначения *p*, *q* относятся к типам `int`, `long`, `double`; а обозначения *P*, *Q* — к типам `Int`, `Long`, `Double`

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода
`BooleanSupplier`	Отсутствует	`boolean`	`getAsBoolean`
P`Supplier`	Отсутствует	*p*	`getAs`*P*
P`Consumer`	*p*	`void`	`accept`

Окончание табл. 3.2

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода
`ObjPConsumer<T>`	`T, p`	`void`	`accept`
`PFunction<T>`	`p`	`T`	`apply`
`PToQFunction`	`T`	`q`	`applyAsQ`
`ToPFunction<T>`	`T`	`T`	`applyAsP`
`ToPBiFunction<T, U>`	`T, U`	`p`	`applyAsP`
`PUnaryOperator`	`p`	`p`	`applyAsP`
`PBinaryOperator`	`p, p`	`p`	`applyAsP`
`PPredicate`	`p`	`boolean`	`test`

3.6.3. Реализация собственных функциональных интерфейсов

Нередко бывают случаи, когда стандартные функциональные интерфейсы не подходят для решения конкретной задачи. В таком случае придется реализовать собственный функциональный интерфейс.

Допустим, что требуется заполнить изображение образцами цвета, где пользователь предоставляет функцию, возвращающую цвет каждого пикселя. Для преобразования `(int, int) -> Color` отсутствует стандартный тип. В таком случае можно было бы воспользоваться функциональным интерфейсом `BiFunction<Integer, Integer, Color>`, но это подразумевает автоупаковку.

В данном случае целесообразно определить новый интерфейс следующим образом:

```
@FunctionalInterface
public interface PixelFunction {
  Color apply(int x, int y);
}
```

НА ЗАМЕТКУ. Функциональные интерфейсы следует помечать аннотацией **`@FunctionalInterface`**. Это дает следующие преимущества. Во-первых, компилятор проверяет, что аннотируемый элемент является интерфейсом с единственным абстрактным методом. И во-вторых, страница документирующих комментариев включает в себя пояснение, что объявляемый интерфейс является функциональным.

А теперь можно реализовать метод следующим образом:

```
BufferedImage createImage(int width, int height, PixelFunction f) {
  BufferedImage image = new BufferedImage(width, height,
          BufferedImage.TYPE_INT_RGB);
  for (int x = 0; x < width; x++)
  for (int y = 0; y < height; y++) {
    Color color = f.apply(x, y);
    image.setRGB(x, y, color.getRGB());
  }
  return image;
}
```

При вызове этого метода предоставляется лямбда-выражение, возвращающее значение цвета, соответствующее двум целочисленным значениям, как показано ниже.

```
BufferedImage frenchFlag = createImage(150, 100,
  (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

3.7. Область действия лямбда-выражений и переменных

В последующих разделах поясняется, каким образом переменные действуют в лямбда-выражениях. Материал этих разделов носит отчасти формальный характер, но он очень важен для правильного понимания принципа действия лямбда-выражений.

3.7.1. Область действия лямбда-выражения

Тело лямбда-выражения имеет *ту же самую* область действия, что и вложенный блок кода. В этой области действия соблюдаются те же самые правила для разрешения конфликтов и сокрытия имен. В теле лямбда-выражения *не* допускается объявлять параметр или локальную переменную с таким же именем, как и у локальной переменной:

```
int first = 0;
Comparator<String> comp =
    (first, second) -> first.length() - second.length();
// ОШИБКА: переменная first уже определена!
```

В теле метода нельзя иметь две локальные переменные с одинаковым именем. Следовательно, такие переменные нельзя внедрить и в лямбда-выражении.

Как другое следствие из правила "одной и той же области действия", ключевое слово `this` в лямбда-выражении обозначает параметр `this` метода, создающего лямбда-выражение. Рассмотрим в качестве примера следующий фрагмент кода:

```
public class Application() {
  public void doWork() {
    Runnable runner =
              () -> { ...; System.out.println(this.toString()); ... };
  ...
  }
}
```

В выражении `this.toString()` вызывается метод `toString()` для объекта типа `Application`, а *не* экземпляра типа `Runnable`. В применении ключевого слова `this` в лямбда-выражении нет ничего особенного. Область действия лямбда-выражения вложена в тело метода `doWork()`, а ссылка `this` имеет одно и то же назначение повсюду в этом методе.

3.7.2. Доступ к переменным из объемлющей области действия

Нередко в лямбда-выражении требуется доступ к переменным из объемлющего метода или класса. Рассмотрим в качестве примера следующий фрагмент кода:

```
public static void repeatMessage(String text, int count) {
  Runnable r = () -> {
    for (int i = 0; i < count; i++) {
      System.out.println(text);
    }
  };
  new Thread(r).start();
}
```

Обратите внимание на доступ из лямбда-выражения к переменным параметров, определяемым в объемлющей области действия, а не в самом лямбда-выражении. Рассмотрим далее следующий вызов:

```
repeatMessage("Hello", 1000); // выводит слово Hello 1000 раз
                              // в отдельном потоке исполнения
```

А теперь обратите внимание на переменные `count` и `text` в лямбда-выражении. Если хорошенько подумать, то можно прийти к выводу, что здесь происходит нечто не совсем обычное. Код лямбда-выражения может быть выполнен спустя немало времени после возврата из вызванного метода `repeatMessage()`, когда переменные параметров больше не существуют. Каким же образом переменные `text` и `count` сохраняются до момента выполнения лямбда-выражения?

Чтобы понять происходящее, нужно уточнить представление о лямбда-выражении. Лямбда-выражение имеет следующие три составляющие:

1. Блок кода.
2. Параметры.
3. Значения трех *свободных* переменных, т.е. таких переменных, которые не являются параметрами и не определены в коде.

В рассматриваемом здесь примере лямбда-выражение содержит две свободные переменные: `text` и `count`. В структуре данных, представляющей лямбда-выражение, должны храниться значения этих переменных (в приведенном выше примере — символьная строка `"Hello"` и число `1000`). В таком случае говорят, что эти значения *захвачены* лямбда-выражением. (Механизм захвата значений зависит от конкретной реализации. Например, лямбда-выражение можно преобразовать в объект единственным методом, чтобы скопировать значения свободных переменных в переменные экземпляра этого объекта.)

НА ЗАМЕТКУ. Формально блок кода вместе со значениями свободных переменных называется *замыканием*. В языке Java лямбда-выражения служат в качестве замыканий.

Как видите, лямбда-выражение может захватывать значение переменной из объемлющей области действия. Но для того чтобы захваченное значение было вполне определено, существует следующее важное ограничение: в лямбда-выражении можно ссылаться только на те переменные, значения которых не изменяются. Иногда это ограничение поясняется таким образом: лямбда-выражения захватывают значения, а не переменные. Так, компиляция следующего фрагмента кода приведет к ошибке:

```
for (int i = 0; i < n; i++) {
  new Thread(() -> System.out.println(i)).start();
  // ОШИБКА: захватить переменную i нельзя
}
```

В приведенном выше лямбда-выражении предпринимается попытка захватить переменную `i`, но это недопустимо, поскольку переменная `i` изменяется, а единственное значение для захвата отсутствует. По упомянутому выше правилу в лямбда-выражении могут быть доступны только локальные переменные из объемлющей области действия, называемые в данном случае *действительно конечными*. Действительно конечная переменная вообще не изменяется. Она является конечной или может быть объявлена как `final`.

НА ЗАМЕТКУ. Это же правило распространяется и на переменные, захватываемые локальными внутренними классами (см. далее раздел 3.9). В прошлом это правило носило более жесткий, драконовский характер, поскольку требовало непременно объявлять захватываемые переменные как **`final`**. Но теперь это не требуется.

НА ЗАМЕТКУ. Переменная расширенного цикла **`for`** является действительно конечной, поскольку ее область действия распространяется на единственный шаг цикла. Так, следующий фрагмент кода вполне допустим:

```
for (String arg : args) {
  new Thread(() -> System.out.println(arg)).start();
  // Захватить переменную arg допустимо!
}
```

Новая переменная **`arg`** создается на каждом шаге цикла и присваивается следующему значению из массива **`args`**. С другой стороны, область действия переменной **`i`** в предыдущем примере распространяется на весь цикл.

Как следствие из правила "действительно конечных переменных", в лямбда-выражении нельзя изменить ни одну из захватываемых переменных. Ниже приведен характерный тому пример.

```
public static void repeatMessage(String text, int count, int threads) {
  Runnable r = () -> {
    while (count > 0) {
      count--; // ОШИБКА: изменить захваченную переменную нельзя!
      System.out.println(text);
    }
  };
  for (int i = 0; i < threads; i++) new Thread(r).start();
}
```

НА ЗАМЕТКУ. Не следует полагаться на компилятор, чтобы перехватывать ошибки параллельного доступа. Запрет на изменение распространяется только на локальные переменные. Если же переменная **`count`** является переменной экземпляра или статической переменной из объемлющего класса, то компилятор не выдаст сообщение об ошибке, даже если результат не определен.

На самом деле это даже хорошо. Так, если переменная count обновляется одновременно в двух потоках исполнения, ее значение не определено, как поясняется в главе 10.

ВНИМАНИЕ. Проверку неуместных изменений можно обойти, используя массив единичной длины следующим образом:

```
int[] counter = new int[1];
button.setOnAction(event -> counter[0]++);
```

Переменная **counter** является действительно конечной. Она вообще не изменяется, поскольку всегда ссылается на один и тот же массив. Следовательно, она доступна в лямбда-выражении.

Безусловно, подобного рода код не является потокобезопасным и поэтому малопригодным, разве что для обратного вызова в однопоточном пользовательском интерфейсе. О том, как реализуется потокобезопасный общий счетчик, речь пойдет в главе 10.

3.8. Функции высшего порядка

В языках функциональнного программирования функциям принадлежит господствующее положение. Подобно методам, получающим параметры и возвращающим разнотипные значения, функции могут принимать аргументы и возвращать соответствующие значения. А те функции, которые обрабатывают или возвращают другие функции, называются *функциями высшего порядка*. Это, на первый взгляд, абстрактное понятие находит очень удобное применение на практике. И хотя Java не является языком функционального программирования, поскольку в нем применяются функциональные интерфейсы, тем не менее, основной принцип остается прежним. В последующих разделах будут продемонстрированы некоторые примеры применения функций высшего порядка в интерфейсе Comparator.

3.8.1. Методы, возвращающие функции

Допустим, что в одних случаях требуется отсортировать массив символьных строк в возрастающем порядке, а в других — в убывающем порядке. С этой целью можно создать метод, получающий нужный компаратор следующим образом:

```
public static Comparator<String> compareInDirecton(int direction) {
  return (x, y) -> direction * x.compareTo(y);
}
```

В результате вызова compareInDirection(1) получается компаратор по возрастанию, а в результате вызова compareInDirection(-1) — компаратор по убыванию. Полученный результат может быть передан другому методу, ожидающему подобный интерфейс, например, методу Arrays.sort():

```
Arrays.sort(friends, compareInDirection(-1));
```

В общем, можно без всякого стеснения писать методы, производящие функции (а формально — экземпляры классов, реализующих функциональный интерфейс). Это удобно для формирования специальных функций, передаваемых методам с помощью функциональных интерфейсов.

3.8.2. Методы, изменяющие функции

В предыдущем разделе был продемонстрирован метод, возвращающий компаратор символьных строк по возрастанию или убыванию. Этот принцип можно обобщить, обратив любой компаратор следующим образом:

```
public static Comparator<String> reverse(Comparator<String> comp) {
  return (x, y) -> comp.compare(x, y);
}
```

Этот метод оперирует функциями. Он получает исходную функцию и возвращает видоизмененную. Так, если требуется отсортировать символьные строки в убывающем порядке с учетом регистра, достаточно сделать следующий вызов:

```
reverse(String::compareToIgnoreCase)
```

НА ЗАМЕТКУ. У интерфейса **`Comparator`** имеется метод по умолчанию **`reversed()`**, обращающий заданный компаратор точно таким же образом.

3.8.3. Методы из интерфейса `Comparator`

В интерфейсе `Comparator` имеется целый ряд полезных статических методов, формирующих компараторы в виде функций высшего порядка. Так, метод `comparing()` принимает в качестве параметра функцию извлечения ключа, преобразующую тип **`T`** в сравниваемый тип (например, **`String`**). Эта функция применяется к сравниваемым объектам, а сравнение делается по возвращаемым ключам. Допустим, что имеется массив объектов типа `Person`. Их можно отсортировать по именам лиц следующим образом:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

Компараторы можно связывать в цепочку методом `thenComparing()`, чтобы решать сложные задачи сравнения. Так, если в приведенном ниже примере кода сравниваемые лица имеют одинаковые фамилии, то применяется второй компаратор.

```
Arrays.sort(people, Comparator
    .comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

Имеется несколько разновидностей этих методов. В частности, можно указать компаратор для сравнения по ключам, извлекаемым методами `comparing()` и `thenComparing()`. В следующем примере кода лица сортируются по длине их имен:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
          (s, t) -> s.length() - t.length()));
```

Более того, у методов `comparing()` и `thenComparing()` имеются варианты, в которых исключается упаковка значений типа `int`, `long` или `double`. В качестве примера ниже приведен простейший способ отсортировать лица по длине их имен.

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

Если функция, извлекающая ключ, может возвратить пустое значение null, то можно вызвать методы nullsFirst() и nullsLast(). Эти статические методы принимают в качестве параметра существующий компаратор и видоизменяют его таким образом, чтобы он не генерировал исключение, если встречаются пустые значения null, но сортировал их как меньшие и большие обычных значений. Допустим, что метод getMiddleName() возвращает пустое значение null, если у лица отсутствует второе имя. В таком случае можно сделать вызов Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...)).

Методу nullsLast() требуется компаратор. В данном случае это должен быть компаратор, сравнивающий две символьные строки. Метод naturalOrder() получает компаратор для любого класса, реализующего интерфейс Comparable. Ниже приведен полный вызов для сортировки потенциально пустых вторых имен. Чтобы сделать приведенное ниже выражение более удобочитаемым, применяется статический импорт из пакета java.util.Comparator.*. Следует также иметь в виду, что тип naturalOrder выводится, а статический метод reverseOrder() обращает естественный порядок сортировки.

```
Arrays.sort(people, comparing(Person::getMiddleName,
            nullsFirst(naturalOrder())));
```

3.9. Локальные внутренние классы

Задолго до появления лямбда-выражений в Java имелся механизм для краткого определения классов, реализующих (функциональный или иной) интерфейс. Для функциональных интерфейсов следует определенно пользоваться лямбда-выражениями, но иногда может потребоваться краткая форма интерфейса, не являющегося функциональным. Кроме того, классические конструкции нередко встречаются в унаследованном коде.

3.9.1. Локальные классы

Класс можно определить в теле метода. Такой класс называется *локальным*. Такой способ подходит для классов, преследующих тактические цели. Подобное часто происходит, когда класс реализует интерфейс, а код, вызывающий метод, интересует только интерфейс, но не класс. Рассмотрим в качестве примера следующий метод, формирующий бесконечную последовательность случайных целых чисел в заданных пределах:

```
public static IntSequence randomInts(int low, int high)
```

Этот метод должен возвращать объект некоторого класса, реализующего интерфейс IntSequence. Вызывающий код не интересует сам класс, и поэтому он может быть объявлен в теле метода следующим образом:

```
private static Random generator = new Random();

public static IntSequence randomInts(int low, int high) {
  class RandomSequence implements IntSequence {
    public int next() { return low + generator.nextInt(high - low + 1); }
    public boolean hasNext() { return true; }
  }

  return new RandomSequence();
}
```

НА ЗАМЕТКУ. Локальный класс не объявляется как **public** или **private**, поскольку он вообще недоступен за пределами метода.

Создание локального класса дает два преимущества. Во-первых, его имя скрыто в области действия метода. И во-вторых, методы локального класса доступны в объемлющей области действия подобно переменным в лямбда-выражении.

В рассматриваемом здесь примере метод `next()` захватывает три переменные: `low`, `high` и `generator`. Если же попробовать создать вложенный класс `RandomInt`, то придется предоставить явный конструктор, получающий эти значения и сохраняющий их в переменных экземпляра (см. упражнение 15 в конце главы).

3.9.2. Анонимные классы

В примере кода из предыдущего раздела имя класса `RandomSequence` использовалось лишь однажды для формирования возвращаемого значения. В таком случае класс можно сделать *анонимным*:

```
public static IntSequence randomInts(int low, int high) {
  return new IntSequence() {
    public int next() { return low + generator.nextInt(high - low + 1); }
    public boolean hasNext() { return true; }
  }
}
```

Приведенное ниже выражение означает следующее: определить класс, реализующий интерфейс, имеющий заданные методы, и построить один объект данного класса.

```
new Интерфейс() { методы }
```

НА ЗАМЕТКУ. Как всегда, круглые скобки в выражении **new** обозначают аргументы конструктора. В данном случае вызывается конструктор анонимного класса по умолчанию.

До появления в Java лямбда-выражений анонимные внутренние классы считались самым кратким синтаксисом для предоставления исполняемых потоков, компараторов и прочих функциональных объектов. Подобные объекты нередко встречаются в унаследованном коде.

Ныне анонимные классы нужны лишь для предоставления двух или больше методов, как в предыдущем примере кода. Если же у интерфейса `IntSequence` имеется

метод по умолчанию hasNext(), как в упражнении 15 в конце этой главы, то вместо анонимного класса можно воспользоваться лямбда-выражением:

```
public static IntSequence randomInts(int low, int high) {
  return () -> low + generator.nextInt(high - low + 1);
}
```

Упражнения

1. Предоставьте интерфейс Measurable с методом double getMeasure(), измеряющим объект определенным образом. Создайте класс Employee, реализующий интерфейс Measurable. Предоставьте метод double average(Measurable[] objects), вычисляющий среднюю меру. Воспользуйтесь им для расчета средней зарплаты в массиве работников.
2. Продолжите предыдущее упражнение, предоставив метод Measurable largest(Measurable[] objects). Воспользуйтесь им, чтобы выяснить имя работника с самой высокой зарплатой. Зачем требуется приведение типов?
3. Каковы все супертипы для типа String, Scanner или ImageOutputStream? Следует иметь в виду, что у каждого типа имеется свой супертип. Класс или интерфейс без явно объявленного супертипа имеет супертип Object.
4. Реализуйте статический метод of() из интерфейса IntSequence, возвращающий последовательность из передаваемых ему аргументов. Например, в результате вызова метода IntSequence.of(3, 1, 4, 1, 5, 9) возвращается последовательность из шести значений. В качестве дополнительного задания организуйте возврат экземпляра анонимного внутреннего класса.
5. Реализуйте статический метод constant() из интерфейса IntSequence, возвращающий бесконечную последовательность констант. Например, в результате вызова IntSequence.constant(1) возвращается бесконечная последовательность 1 1 1 . . . . В качестве дополнительного задания сделайте то же самое с помощью лямбда-выражения.
6. В этом упражнении вам предстоит выяснить, что произойдет, если ввести метод в интерфейс. В версии Java 7 создайте класс DigitSequence, реализующий интерфейс Iterator<Integer>, а не IntSequence. Предоставьте методы hasNext(), next() и ничего не делающий метод remove(). Напишите программу, выводящую элементы интерфейса. В версии Java 8 интерфейс Iterator был дополнен методом forEachRemaining(). Будет ли ваш код по-прежнему компилироваться при переходе к версии Java 8? Если вы введете свой класс в архивный JAR-файл, не перекомпилировав его, будет ли он нормально действовать в версии Java 8? Что произойдет, если вызвать метод forEachRemaining()? Кроме того, метод remove() стал методом по умолчанию в версии Java 8, генерируя исключение типа UnsupportedOperationException. Что произойдет, если вы вызовете метод remove() для экземпляра своего класса?

7. Реализуйте метод `void luckySort(ArrayList<String> strings, Comparator <String> comp)`, вызывающий метод `Collections.shuffle()` для списочного массива до тех пор, пока элементы этого массива располагаются в возрастающем порядке, задаваемом компаратором.

8. Создайте класс `Greeter`, реализующий интерфейс `Runnable`, метод которого `run()` выводит `n` раз сообщение `"Hello, " + target`, где `n` и `target` — параметры, устанавливаемые в конструкторе. Получите два экземпляра этого класса с разными сообщениями и выполните их параллельно в двух потоках.

9. Реализуйте следующие методы:

```
public static void runTogether(Runnable... tasks)
public static void runInOrder(Runnable... tasks)
```

   Первый метод должен выполнять каждую задачу в отдельном потоке и возвращать полученный результат, а второй метод — все методы в текущем потоке и возвращать полученный результат по завершении последнего метода.

10. Используя методы `listFiles(FileFilter)` и `isDirectory` из класса `java.io.File`, напишите метод, возвращающий все подкаталоги из заданного каталога. Воспользуйтесь для этой цели лямбда-выражением вместо объекта типа `FileFilter`. Сделайте то же самое, используя ссылку на метод и анонимный внутренний класс.

11. Используя метод `list(FilenameFilter)` из класса `java.io.File`, напишите метод, возвращающий все файлы из заданного каталога с указанным расширением. Воспользуйтесь для этой цели лямбда-выражением вместо объекта типа `FilenameFilter`. Какая переменная из объемлющей области действия захватывается лямбда-выражением?

12. Если задан массив объектов типа `File`, отсортируйте его таким образом, чтобы каталоги следовали перед файлами, а в каждой группе отсортируйте элементы по пути к ним. Воспользуйтесь лямбда-выражением, чтобы указать компаратор типа `Comparator`.

13. Напишите метод, принимающий массив экземпляров типа `Runnable` и возвращающий экземпляр типа `Runnable`, метод которого `run()` выполняет их по порядку. Организуйте возврат лямбда-выражения.

14. Организуйте вызов метода `Arrays.sort()`, сортирующего работников сначала по зарплате, а затем по имени. Воспользуйтесь для этой цели методом `Comparator.thenComparing()`. Затем организуйте сортировку в обратном порядке.

15. Реализуйте локальный класс `RandomSequence`, упоминавшийся в разделе 3.9.1, как вложенный класс за пределами метода `randomInts()`.

Глава 4

Наследование и рефлексия

В этой главе...

- 4.1. Расширение класса
- 4.2. Всеобъемлющий суперкласс `Object`
- 4.3. Перечисления
- 4.4. Динамическая идентификация типов: сведения и ресурсы
- 4.5. Рефлексия
- Упражнения

В предыдущих главах были представлены классы и интерфейсы. А в этой главе речь пойдет о наследовании — еще одном основополагающем понятии объектно-ориентированного программирования. Наследование — это процесс создания новых классов на основании уже имеющихся. При наследовании из существующего класса повторно используются (или наследуются) его методы, а кроме того, имеется возможность вводить новые методы и поля.

НА ЗАМЕТКУ. Переменные экземпляра и статические переменные совместно называются *полями*, а поля, методы, вложенные классы и интерфейсы в классе — *членами* этого класса.

В этой главе рассматривается также рефлексия — способность получать дополнительные сведения о классах и их членах по мере выполнения программы. Рефлексия является эффективным, но, безусловно, сложным средством. Рефлексия больше интересует разработчиков инструментальных средств, чем прикладных программистов, и поэтому вам, возможно, придется впоследствии вернуться к той части этой главы, где обсуждается рефлексия.

Основные положения этой главы приведены ниже.

1. Подкласс может наследовать или переопределять методы из суперкласса.
2. Ключевое слово `super` служит для вызова метода или конструктора из суперкласса.
3. Метод, объявленный как `final`, не может быть переопределен, а класс, объявленный как `final`, — расширен.
4. Метод, объявленный как `abstract`, не имеет своей реализации, а класс, объявленный как `abstract`, — своего экземпляра.
5. Защищенный (`protected`) член подкласса доступен в методе этого подкласса, но только в том случае, если он применяется к объектам того же самого подкласса.
6. Каждый класс является подклассом, производным от класса `Object`, предоставляющего методы `toString()`, `equals()`, `hashCode()` и `clone()`.
7. Каждый перечислимый тип является подклассом, производным от класса `Enum`, предоставляющего методы `toString()`, `valueOf()` и `compareTo()`.
8. Класс `Class` предоставляет сведения о типе данных, который может быть в Java классом, массивом, интерфейсом, примитивным типом или типом `void`.
9. Объект типа `Class` можно использовать для загрузки ресурсов, размещаемых наряду с файлами классов.
10. Классы можно загружать не только по пути к их файлам, но и из других мест, используя загрузчик классов.
11. Библиотека рефлексии позволяет обнаруживать в программах члены произвольных объектов, получать доступ к переменным и вызывать методы.
12. Объекты-заместители динамически реализуют произвольные интерфейсы, направляя все вызовы методов соответствующему обработчику.

4.1. Расширение класса

Обратимся снова к примеру класса Employee, обсуждавшегося в главе 2. Допустим, что отношение к руководителям компании иное, чем к остальным ее работникам. Безусловно, руководители во многом такие же работники, как и все остальные. Руководителям и остальным работникам платят зарплату. Но если работники выполняют свои обязанности, получая за это определенную плату, то руководители получают еще и премии, если они делают то, что от них требуется. Подобная ситуация может быть смоделирована путем наследования.

4.1.1. Суперклассы и подклассы

Определим новый класс Manager, сохраняющий некоторые функции класса Employee, но в то же время обозначающий отличия руководителей от остальных работников. Этот класс определяется следующим образом:

```
public class Manager extends Employee {
  дополнительные поля
  дополнительные или переопределяемые методы
}
```

Ключевое слово extends обозначает, что создается новый класс, производный от существующего класса. Существующий класс называется *суперклассом*, тогда как новый класс — *подклассом*. В рассматриваемом здесь примере класс Employee является суперклассом, а класс Manager — подклассом. Следует, однако, иметь в виду, что суперкласс на самом деле ничем не "превосходит" подкласс. Напротив, подклассы обладают большими функциональными возможностями, чем их суперклассы. Терминология подклассов и суперклассов заимствована из теории множеств.

4.1.2. Определение и наследование методов из суперкласса

В рассматриваемом здесь классе Manager имеется новая переменная экземпляра для хранения суммы премии и новый метод для ее установки:

```
public class Manager extends Employee {
  private double bonus;
  ...
  public void setBonus(double bonus) {
    this.bonus = bonus;
  }
}
```

Имея в своем распоряжении объект типа Manager, можно, конечно, применить метод setBonus(), а также незакрытые методы из класса Employee, как показано ниже. Эти методы *наследуются*.

```
Manager boss = new Manager(...);
boss.setBonus(10000); // Определен в суперклассе
boss.raiseSalary(5); // Наследуется из суперкласса
```

4.1.3. Переопределение методов

Иногда метод из суперкласса приходится видоизменять в подклассе. Допустим, что метод `getSalary()` должен сообщать о совокупной зарплате работника. В таком случае наследуемого метода оказывается недостаточно для класса `Manager`. Поэтому данный метод придется *переопределить* таким образом, чтобы он возвращал сумму основной зарплаты и премии. Ниже показано, как это делается.

```
public class Manager extends Employee {
  ...
  public double getSalary() { // Переопределяет метод из суперкласса
    return super.getSalary() + bonus;
  }
}
```

Этот метод сначала вызывает метод из суперкласса, получающий основную зарплату, а затем добавляет к ней премию. Следует, однако, иметь в виду, что методу из подкласса недоступны непосредственно переменные экземпляра из суперкласса. Именно поэтому в методе `Manager.getSalary()` вызывается открытый метод `Employee.getSalary()`. А ключевое слово `super` служит для вызова метода из суперкласса.

НА ЗАМЕТКУ. В отличие от ключевого слова **`this`**, ключевое слово **`super`** является не ссылкой на объект, а директивой для действия в обход динамического поиска методов (см. далее раздел 4.1.5) и вызова конкретного метода.

При переопределении метода вызывать метод из суперкласса не требуется, хотя это часто делается. Переопределяя метод, следует точно соблюдать соответствие типов параметров. Допустим, что в классе `Employee` имеется следующий метод:

```
public boolean worksFor(Employee supervisor)
```

Если переопределить этот метод в классе `Manager`, то изменить тип его параметра нельзя, несмотря на то, что ни один руководитель не отчитывается перед подчиненными. Допустим также, что в классе `Manager` определен аналогичный метод:

```
public class Manager extends Employee {
  ...
  public boolean worksFor(Manager supervisor) {
    ...
  }
}
```

Это просто новый метод, и теперь в классе `Manager` имеются два отдельных метода `worksFor()`. Чтобы уберечься от подобного рода ошибок, методы из суперкласса, которые предполагается переопределить в подклассе, рекомендуется пометить аннотацией `@Override`, как показано ниже. Если по ошибке будет определен аналогичный новый метод, компилятор выдаст сообщение о подобной ошибке.

```
@Override public boolean worksFor(Employee supervisor)
```

При переопределении метода в подклассе допускается изменять возвращаемый тип. (Формально выражаясь, разрешаются *ковариантные возвращаемые типы*.) Так, если в классе Employee имеется следующий метод:

```
public Employee getSupervisor()
```

его можно переопределить в классе Manager таким методом:

```
@Override public Manager getSupervisor()
```

ВНИМАНИЕ. Когда метод переопределяется в подклассе, он должен быть доступен *хотя бы* как метод из суперкласса. Так, если метод из суперкласса является открытым, то и метод, переопределяемый в подклассе, также должен быть открытым. При переопределении метода в подклассе нередко допускают ошибку, неумышленно пропуская модификатор доступа **public**. В таком случае компилятор выдаст предупреждение об ослаблении привилегий доступа.

4.14. Построение подкласса

А теперь нужно снабдить класс Manager конструктором. Конструктору этого класса недоступны закрытые переменные экземпляра из класса Employee, и поэтому они должны быть инициализированы через конструктор его суперкласса:

```
public Manager(String name, double salary) {
  super(name, salary);
  bonus = 0;
}
```

В данном случае ключевое слово super обозначает вызов конструктора из суперкласса Employee с аргументами name и salary. При этом вызов конструктора из суперкласса должен быть *первым оператором* в теле конструктора подкласса. Если же опустить вызов конструктора из суперкласса, то у суперкласса должен быть конструктор без аргументов, который вызывается неявно.

4.1.5. Присваивания в суперклассе

Вполне допустимо присваивать объект из подкласса переменной, тип которой относится к суперклассу, как в следующем примере кода:

```
Manager boss = new Manager(...);
Employee empl = boss; // Присваивание переменной из суперкласса допустимо!
```

А теперь рассмотрим, что произойдет, если вызвать метод для переменной из суперкласса следующим образом:

```
double salary = empl.getSalary();
```

Вызов метода Manager.getSalary() происходит, несмотря на то, что переменная empl относится к типу Employee. Во время этого вызова виртуальная машина анализирует конкретный класс объекта и находит вариант его метода. Этот процесс называется *динамическим поиском методов*.

А зачем вообще присваивать объект типа Manager переменной типа Employee? Это дает возможность написать код, пригодный для обращения ко *всем* работникам, будь то руководители или подчиненные, как показано ниже.

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...); // Присваивание переменной из суперкласса допустимо!
staff[2] = new Janitor(...);
...
double sum = 0;
for (Employee empl : staff)
  sum += empl.getSalary();
```

Благодаря динамическому поиску методов вызов empl.getSalary() влечет за собой обращение к методу getSalary(), относящемуся к тому объекту, на который ссылается переменная empl. А это может быть ссылка на метод Employee.getSalary(), Manager.getSalary() и т.д.

ВНИМАНИЕ. В языке Java присваивание в суперклассе допускается и для массивов. В частности, массив типа **Manager[]** можно присвоить переменной типа **Employee[]**. (Формально выражаясь, массивы в Java являются *ковариантными*.) Это удобно, но и ненадежно, поскольку может привести к ошибкам соблюдения типов. Рассмотрим следующий пример кода:

```
Manager[] bosses = new Manager[10];
Employee[] empls = bosses; // Допустимо в Java!
empls[0] = new Employee(...); // ОШИБКА при выполнении!
```

Компилятор правильно воспринимает последний оператор, поскольку в Java обычно допускается хранить объекты типа **Employee** в массиве типа **Employee[]**. Но в данном случае ссылки **empls** и **bosses** делаются на один и тот же массив супертипа **Manager[]**, в котором нельзя хранить объекты подтипа **Employee**. Такую ошибку можно перехватить только во время выполнения, когда виртуальная машины генерирует исключение типа **ArrayStoreException**.

4.1.6. Приведение типов

В предыдущем разделе было показано, каким образом переменная empl типа Employee может ссылаться на объекты суперкласса Employee, подкласса Manager или другого подкласса, производного от класса Employee. Это удобно для написания кода, в котором происходит обращение к объектам из разных классов. Но у такого подхода имеется следующий недостаток: вызывать можно только те методы, которые относятся к суперклассу. В качестве примера рассмотрим следующий фрагмент кода:

```
Employee empl = new Manager(...);
empl.setBonus(10000); // ОШИБКА при компиляции!
```

Этот код приводит к ошибке при компиляции, несмотря на то, что он мог бы привести к удачному исходу во время выполнения. Дело в том, что компилятор проверяет, что вызываться должны только те методы, которые существуют для типа получателя. В данном примере переменная empl относится к типу Employee, т.е. к классу, где отсутствует метод setBonus().

Как и в интерфейсах, для приведения типа ссылки из суперкласса на подкласс можно воспользоваться операцией instanceof следующим образом:

```
if (empl instanceof Manager) {
  Manager mgr = (Manager) empl;
  mgr.setBonus(10000);
}
```

4.1.7. Конечные методы и классы

Если метод объявляется как `final`, его уже нельзя переопределить в подклассе. Ниже приведен пример объявления конечного метода в суперклассе.

```
public class Employee {
  ...
  public final String getName() {
    return name;
  }
}
```

Характерным примером конечного метода в прикладном программном интерфейсе Java API служит метод `getClass()` из класса `Object`, рассматриваемого далее в разделе 4.4.1. На первый взгляд, неразумно позволять объектам сообщать неверные сведения о классе, к которому они относятся, и поэтому данный метод не подлежит изменению.

Некоторые программисты считают, что ключевое слово `final` способствует эффективности прикладного кода. Возможно, это и было именно так на ранней стадии развития Java, но не теперь. Современные виртуальные машины эмпирически "встраивают" простые методы вроде упомянутого выше метода `getName()`, даже если они не объявлены как `final`. И только в редких случаях, когда загружается подкласс, в котором такой метод переопределяется, встраивание отменяется.

Одни программисты считают, что большинство методов должно быть объявлено в классе как `final`, кроме методов, специально предназначенных для переопределения. А другие программисты усматривают в этом драконовское правило, поскольку оно препятствует даже безвредному переопределению методов, например, для целей протоколирования или отладки.

Если вам требуется не допустить формирование кем-нибудь другим подкласса, производного от одного из ваших классов, воспользуйтесь модификатором доступа `final` в определении своего класса, чтобы указать на его конечный характер. В качестве примера ниже показано, как воспрепятствовать другим осуществить подклассификацию класса `Executive`. В прикладном программном интерфейсе Java API имеется немало конечных классов, в том числе `String`, `LocalTime` и `URL`.

```
public final class Executive extends Manager {
  ...
}
```

4.1.8. Абстрактные методы и классы

В классе можно определить метод без его реализации, вынудив реализовать его в подклассах. Такой метод и содержащий его класс называются *абстрактными* и должны быть помечены модификатором доступа `abstract`. Как правило, это делается для создания самых общих классов, как в следующем примере кода:

```
public abstract class Person {
  private String name;

  public Person(String name) { this.name = name; }
  public final String getName() { return name; }

  public abstract int getId();
}
```

Любой класс, расширяющий класс `Person`, должен предоставлять реализацию метода `getId()` или же должен быть сам объявлен как `abstract`. Следует, однако, иметь в виду, что абстрактный класс может содержать и неабстрактные методы вроде метода `getName()` из предыдущего примера.

НА ЗАМЕТКУ. В отличие от интерфейса, абстрактный класс может иметь переменные экземпляра и конструкторы.

Получить экземпляр абстрактного класса нельзя. Например, следующий вызов конструктора приведет к ошибке при компиляции:

```
Person p = new Person("Fred"); // ОШИБКА!
```

Но в то же время можно иметь переменную, тип которой относится к абстрактному классу, при условии, что она содержит ссылку на объект конкретного подкласса. Допустим, что класс `Student` объявлен следующим образом:

```
public class Student extends Person {
  private int id;

  public Student(String name, int id) { super(name); this.id = id; }
  public int getId() { return id; }
}
```

В таком случае можно построить объект типа `Student` и присвоить его переменной типа `Person`, как показано ниже.

```
Person p = new Student("Fred", 1729); // Верно, поскольку это
                                      // конкретный подкласс!
```

4.1.9. Защищенный доступ

Иногда требуется ограничить метод только подклассами, а еще реже — разрешить методам из подкласса доступ к переменной экземпляра суперкласса. Для этого достаточно объявить данный член класса как `protected`.

ВНИМАНИЕ. В языке Java ключевое слово **`protected`** обеспечивает доступ на уровне пакета и защищает *только* от доступа из других пакетов.

Допустим, что переменная экземпляра `salary` объявляется в суперклассе `Employee` как `protected`, а не как `private`. Ниже показано, как это делается.

```
package com.horstmann.employees;

public class Employee {
  protected double salary;
  ...
}
```

Это поле доступно всем классам из того же самого пакета, где находится класс Employee. А теперь рассмотрим следующий подкласс из другого пакета:

```
package com.horstmann.managers;

import com.horstmann.employees.Employee;

public class Manager extends Employee {
  ...
  public double getSalary() {
    return salary + bonus; // Верно для доступа к защищенной переменной salary
  }
}
```

Методы из класса Manager могут обращаться только к содержимому поля объектов типа Manager, но не других объектов типа Employee. Подобное ограничение накладывается для того, чтобы не нарушить механизм защиты, формируя подклассы лишь для получения доступа к защищенным языковым средствам.

Разумеется, защищенными полями следует пользоваться осторожно. Если они предоставляются, то от них нельзя избавиться, не нарушая классы, которые пользуются ими.

Большее распространение получили защищенные методы и конструкторы. Например, метод clone() из класса Object защищен, поскольку пользоваться им непросто (см. далее раздел 4.2.4).

4.1.10. Анонимные подклассы

Допускается иметь не только анонимный класс, реализующий интерфейс, но и анонимный класс, расширяющий суперкласс, как показано ниже. Это очень удобно для целей отладки.

```
ArrayList<String> names = new ArrayList<String>(100) {
  public void add(int index, String element) {
    super.add(index, element);
    System.out.printf("Adding %s at %d\n", element, index);
  }
};
```

Аргументы в круглых скобках передаются конструктору суперкласса, следуя имени последнего. В данном примере строится анонимный подкласс, производный от класса ArrayList<String>, где переопределяется метод add(). Его экземпляр получается с первоначальной емкостью списочного массива, равной 100.

В специальном приеме, называемом *инициализацией в двойных фигурных скобках*, синтаксис внутреннего класса используется довольно необычным образом. Допустим, что требуется построить списочный массив и передать ему метод, как показано ниже.

```
ArrayList<String> friends = new ArrayList<>();
friends.add("Harry");
friends.add("Sally");
invite(friends);
```

Если списочный массив больше не потребуется, то его целесообразно сделать анонимным. Но как тогда ввести в него элементы? А вот как:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Sally"); }});
```

Обратите внимание на двойные фигурные скобки. Внешние фигурные скобки служат для создания анонимного подкласса, производного от класса `ArrayList<String>`. А внутренние фигурные скобки обозначают блок инициализации (см. главу 2).

Пользоваться этим специальным приемом не рекомендуется вне обычного для Java контекста. Помимо запутанного синтаксиса, этому приему присущ ряд других недостатков. Он неэффективен, а построенный объект может вести себя необычно при проверках на равенство, хотя это зависит от реализации метода `equals()`.

4.1.11. Наследование и методы по умолчанию

Допустим, что один класс расширяет другой класс и реализует интерфейс, причем и в том и в другом имеется метод с одинаковым именем, как показано ниже.

```
public interface Named {
  default String getName() { return ""; }
}

public class Person {
   ...
  public String getName() { return name; }
}

public class Student extends Person implements Named {
  ...
}
```

В таком случае предпочтение отдается реализации метода в суперклассе над его реализацией в интерфейсе. А подкласс избавляется от необходимости разрешать конфликт имен.

С другой стороны, конфликт имен необходимо разрешить, если один и тот же метод по умолчанию наследуется от двух интерфейсов, как пояснялось в главе 3. Правило, отдающее предпочтение реализации метода в суперклассе, гарантирует совместимость с версией Java 7. Если ввести методы по умолчанию в интерфейс, это никак не повлияет на прежний прикладной код, действовавший до появления методов по умолчанию.

4.1.12. Ссылки на методы типа `super`

Как упоминалось в главе 3, ссылка на метод может принимать следующую форму:

Объект*::*МетодЭкземпляра

В качестве ссылки на объект в этой форме допускается использовать ключевое слово `super`. В таком случае ссылка на метод примет следующую форму:

super::*МетодЭкземпляра*

В этой форме используется ссылка this на адресат для вызова варианта заданного метода из суперкласса. Ниже приведен искусственный пример, демонстрирующий данный механизм. Поток исполнения строится средствами интерфейса Runnable, в методе run() которого вызывается метод work() из суперкласса.

```
public class Worker {
  public void work() {
    for (int i = 0; i < 100; i++) System.out.println("Working");
  }
}

public class ConcurrentWorker extends Greeter {
  public void work() {
    Thread t = new Thread(super::work);
    t.start();
  }
}
```

4.2. Всеобъемлющий суперкласс Object

Каждый класс в Java прямо или косвенно расширяет класс Object. Если у класса явно отсутствует суперкласс, то он неявно расширяет класс Object. Например, следующее объявление класса:

```
public class Employee { ... }
```

равнозначно такому объявлению:

```
public class Employee extends Object { ... }
```

В классе Object определяются методы, применяемые к объекту любого типа в Java (табл. 4.1). Некоторые из этих методов будут рассмотрены в последующих разделах.

Таблица 4.1. Методы из класса **java.lang.Object**

Метод	Описание
String toString()	Возвращает строковое представление данного объекта (по умолчанию — имя класса и хеш-код)
boolean equals(Object *other*)	Возвращает логическое значение **true**, если данный объект считается равным указанному объекту ***other***; а если указанный объект ***other*** принимает пустое значение **null** или же отличается от данного объекта, — логическое значение **false**. По умолчанию оба объекта считаются равными, если они одинаковы. Вместо вызова метода **obj.equals(*other*)** рекомендуется вызвать его вариант **Object.equals(obj, *other*)**, безопасно обрабатывающий пустые значения
int hashCode()	Возвращает хеш-код для данного объекта. У равных объектов должен быть одинаковый хеш-код. Если данный метод не переопределяется, то хеш-код присваивается некоторым образом виртуальной машиной
Class<?> getClass()	Возвращает объект типа **Class**, описывающий класс, к которому этот объект относится

Окончание табл. 4.1

Метод	Описание
`protected Object clone()`	Создает копию данного объекта. По умолчанию копия оказывается неполной
`protected void finalize()`	Этот метод вызывается в том случае, когда данный объект утилизируется сборщиком "мусора". Он подлежит переопределению
`wait()`, `notify()`, `notifyAll()`	См. главу 10

НА ЗАМЕТКУ. Массивы, по существу, являются классами. Следовательно, вполне допустимо преобразовать массив (даже примитивного типа) в ссылку типа `Object`.

4.2.1. Метод `toString()`

В классе `Object` имеется очень важный метод `toString()`, возвращающий строковое представление объекта. Например, в классе `Point` вариант этого метода возвращает символьную строку, аналогичную следующей:

```
java.awt.Point[x=10,y=20]
```

Многие варианты метода `toString()` придерживаются следующего формата: имя класса, после которого следуют переменные экземпляра в квадратных скобках. В качестве примера ниже приведена реализация метода `toString()` в классе `Employee`.

```
public String toString() {
  return getClass().getName() + "[name=" + name
        + ",salary=" + salary + "]";
}
```

Благодаря вызову `getClass().getName()` вместо жесткого кодирования символьной строки `"Employee"` этот метод оказывается вполне пригодным и для подклассов. Так, в подклассе вызов `super.toString()` можно дополнить переменными экземпляра подкласса в отдельных квадратных скобках, как показано ниже.

```
public class Manager extends Employee {
  ...
  public String toString() {
    return super.toString() + "[bonus=" + bonus + "]";
  }
}
```

Всякий раз, когда объект сцепляется с символьной строкой, компилятор автоматически вызывает метод `toString()` для данного объекта, как показано в следующем примере кода:

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
  // Сцепляет данную строку со строковым представлением объекта p,
  // получаемым в результате автоматического вызова p.toString()
```

СОВЕТ. Вместо вызова `x.toString()` можно написать выражение `"" + x`. Это выражение действует даже в том случае, если переменная `x` содержит пустое значение `null` или значение примитивного типа.

В классе `Object` метод `toString()` определяется для вывода имени класса и хеш-кода (см. далее раздел 4.2.3). Например, в результате вызова

```
System.out.println(System.out)
```

выводится результат, аналогичный следующему: `java.io.PrintStream@2f6684`, поскольку разработчик класса `PrintStream` не позаботился о переопределении метода `toString()` в этом классе.

ВНИМАНИЕ. Массивы наследуют метод `toString()` от класса `Object`, но при условии, что тип массива выводится в архаичном формате. Так, если имеется следующий массив:

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

то в результате вызова `primes.toString()` возвращается символьная строка вроде `"[I@1a46e30"`, где префикс `[I` обозначает массив целочисленных значений.

В качестве выхода из этого положения можно сделать вызов `Arrays.toString(primes)`, в результате которого возвращается символьная строка `"[2, 3, 5, 7, 11, 13]"`. Чтобы правильно вывести строковое представление многомерных массивов, т.е. одних массивов, содержащих другие массивы, следует вызвать метод `Arrays.deepToString()`.

4.2.2. Метод `equals()`

В методе `equals()` проверяется, равен ли один объект другому. В реализации этого метода в классе `Object` определяется, являются ли равными ссылки на сравниваемые объекты. И это вполне обосновано по умолчанию. Ведь если два объекта одинаковы, то они должны быть, безусловно, равны. Ничего другого больше не требуется, разве что для некоторых классов. Например, вряд ли стоит сравнивать на равенство два объекта типа `Scanner`.

Метод `equals()` следует переопределять только для проверки объектов на равенство с учетом их состояния. В этом случае два объекта считаются равными, если одинаково их содержимое. Например, в классе `String` метод `equals()` переопределяется, чтобы проверить, состоят ли две сравниваемые строки из одних и тех же символов.

ВНИМАНИЕ. Всякий раз, когда метод `equals()` переопределяется, *необходимо* также предоставить совместимый с ним метод `hashCode()` [см. далее раздел 4.2.3.]

Допустим, что два объекта класса `Item` требуется признать равными, если описания и цены на товары в них одинаковы. Ниже показано, как в этом случае реализовать метод `equals()`.

```
public class Item {
  private String description;
```

```
  private double price;
  ...
  public boolean equals(Object otherObject) {
    // Быстрая проверка объектов на сходство
    if (this == otherObject) return true;
       // возвратить логическое значение false, если параметр
       // принимает пустое значение null
    if (otherObject == null) return false;
      // проверить, относится ли объект otherObject к типу Item
    if (getClass() != otherObject.getClass()) return false;
      // проверить, содержат ли переменные экземпляра одинаковые значения
    Item other = (Item) otherObject;
    return Objects.equals(description, other.description)
          && price == other.price;
  }

  public int hashCode() { ... } // См. раздел 4.2.3
}
```

В методе `equals()` нужно выполнить целый ряд рутинных шагов, перечисленных ниже.

1. Нередко равные объекты одинаковы, и поэтому быстрая проверка на их сходство оказывается весьма экономной.
2. Каждый метод `equals()` должен возвращать логическое значение `false` при сравнении с пустым значением `null`.
3. Реализуемый метод `equals()` переопределяет метод `Object.equals()`, и поэтому его параметр относится к типу `Object`, а значит, его тип нужно привести к конкретному типу, чтобы проанализировать содержимое его переменных экземпляра. Но прежде чем сделать это, следует произвести проверку соответствия типов с помощью класса `getClass()` или операции `instanceof`.
4. И наконец, следует сравнить переменные экземпляра. Для сравнения примитивных типов можно воспользоваться операцией **==**. Но если при сравнении числовых значений типа `double` необходимо принимать во внимание $\pm\infty$ и не число (NaN), то лучше вызвать метод `Double.equals()`. А для сравнения объектов следует вызвать вариант метода `Objects.equals()`, безопасно обрабатывающий пустые значения. Так, в результате вызова `Objects.equals(x, y)` возвращается логическое значение `false`, если параметр `x` принимает пустое значение `null`, тогда как в результате вызова `x.equals(y)` генерируется исключение при том же условии.

СОВЕТ. Если имеются переменные экземпляра, содержащие массивы, следует вызвать метод **`Arrays.equals()`**, чтобы проверить, имеют ли массивы одинаковую длину и равные соответствующие элементы.

Если метод `equals()` определяется в подклассе, то сначала следует вызвать аналогичный метод `equals()` из суперкласса. Если проверка не пройдет, то объекты нельзя считать равными. А если переменные экземпляра из суперкласса равны, то

можно приступить к сравнению переменных экземпляра и в подклассе. Ниже приведен пример определения метода `equals()` в подклассе. Проверка с помощью метода `getClass()` в суперклассе не пройдет, если объект `otherObject` не относится к типу `DiscountedItem`.

```
public class DiscountedItem extends Item {
  private double discount;
  ...
  public boolean equals(Object otherObject) {
    if (!super.equals(otherObject)) return false;
    DiscountedItem other = (DiscountedItem) otherObject;
    return discount == other.discount;
  }

  public int hashCode() { ... }
}
```

Каким же должно быть поведение метода `equals()` при сравнении объектов, относящихся к разным классам? Здесь может возникнуть противоречие. В предыдущем примере метод `equals()` возвращает логическое значение `false`, если объекты не совпадают полностью. Но многие программирующие на Java пользуются вместо этого операцией `instanceof` следующим образом:

```
if (!(otherObject instanceof Item)) return false;
```

Это оставляет возможность для принадлежности объекта `otherObject` подклассу. Например, объект типа `Item` можно сравнить с объектом типа `DiscountedItem`.

Но такое сравнение обычно не срабатывает. Одно из требований к методу `equals()` состоит в том, что он должен действовать *симметрично*. В частности, в результате вызовов `x.equals(y)` и `y.equals(x)` для непустых значений `x` и `y` должно быть возвращено одно и то же значение.

А теперь допустим, что `x` является объектом типа `Item`, а `y` — объектом типа `DiscountedItem`. При вызове `x.equals(y)` скидки на товары не учитываются, а следовательно, они не могут быть учтены и при вызове `y.equals(x)`.

НА ЗАМЕТКУ. В состав прикладного программного интерфейса Java API входит более 150 реализаций метода **`equals()`**, где организованы проверки объектов на равенство с помощью операции **`instanceof`**, вызова метода **`getClass()`**, перехвата исключения типа **`ClassCastException`**, или вообще ничего не делается. Обратитесь к документации на класс **`java.sql.Timestamp`**, где его разработчики отмечают некоторые трудности его применения. В частности, класс **`Timestamp`** наследует от класса **`java.util.Date`**, в методе **`equals()`** которого организована проверка объектов на равенство с помощью операции **`instanceof`**, и поэтому метод **`equals()`** нельзя переопределить таким образом, чтобы он действовал не только симметрично, но и точно.

Проверка объектов на равенство с помощью операции `instanceof` оказывается целесообразной в том случае, если понятие равенства объектов фиксируется в суперклассе и вообще не изменяется в подклассе. В этом случае работников можно сравнить, например, по идентификационным номерам, организовав проверку с помощью операции `instanceof` и объявив метод `equals()` как `final`. Ниже показано, как это делается.

```
public class Employee {
  private int id;
  ...
  public final boolean equals(Object otherObject) {
    if (this == otherObject) return true;
    if (!(otherObject instanceof Employee)) return false;
    Employee other = (Employee) otherObject;
    return id == other.id;
  }

  public int hashCode() { ... }
}
```

4.2.3. Метод hashCode()

Хеш-код представляет собой целочисленное значение, производное от объекта. Хеш-коды должны быть произвольными. Если x и y обозначают два неравных объекта, то, вероятнее всего, разными окажутся и хеш-коды, возвращаемые в результате вызовов x.hashCode() и y.hashCode(). Так, в результате вызова "Mary".hashCode() возвращается хеш-код **2390779**, а в результате вызова "Myra".hashCode() — хеш-код **2413819**.

В классе String применяется следующий алгоритм для вычисления хеш-кода:

```
int hash = 0;
for (int i = 0; i < length(); i++)
  hash = 31 * hash + charAt(i);
```

Методы hashCode() и equals() должны быть *совместимы*. Если возможен вызов x.equals(y), то должна быть возможность и для сравнения x.hashCode() == y.hashCode(). Это, в частности, справедливо для класса String, поскольку строки с одинаковыми символами дают один и тот же хеш-код.

Метод Object.hashCode() производит хеш-код некоторым способом, зависящим от конкретной реализации. Он может быть произведен от местоположения объекта в оперативной памяти или (последовательного или псевдослучайного) числа, кешируемого вместе с объектом, или же и тем, и другим способом. В методе Object.equals() выполняется проверка на равенство объектов, и поэтому важнее всего лишь то, что одинаковые объекты должны иметь один и тот же хеш-код.

Если переопределить метод equals(), то придется переопределить и метод hashCode(), чтобы обеспечить их совместимость. В противном случае объекты, вводимые пользователями класса в хеш-множество или хеш-отображение, могут быть просто утрачены!

Достаточно объединить хеш-коды переменных экземпляра. В качестве примера ниже приведен метод hashCode() из класса Item.

```
class Item {
  ...
  public int hashCode() {
    return Objects.hash(description, price);
  }
}
```

Метод `Objects.hash()` с аргументами переменной длины вычисляет хеш-коды своих аргументов и объединяет их. Этот метод безопасно обрабатывает пустые значения.

Если в классе имеются переменные экземпляра, хранящие массивы, сначала следует вычислить их хеш-коды с помощью статического метода `Arrays.hashCode()`, а затем хеш-код, состоящий из хеш-кодов отдельных элементов массива. Полученный результат передается методу `Objects.hash()`.

ВНИМАНИЕ. В интерфейсе нельзя объявить метод по умолчанию, переопределяющий один из методов в классе **`Object`**. Так, в интерфейсе нельзя определить метод по умолчанию, переопределяющий один из методов **`toString()`**, **`equals()`** или **`hashCode()`**. Как следствие из правила, отдающего предпочтение реализации метода в суперклассе (см. раздел 4.1.1), такой метод не в состоянии соперничать за первенство с методом **`Object.toString()`** или **`Object.equals()`**.

4.2.4. Клонирование объектов

В предыдущих разделах были продемонстрированы три наиболее употребительных метода из класса `Object`, а именно: `toString()`, `equals()` и `hashCode()`. А в этом разделе будет показано, как определяется метод `clone()` из этого же класса. Как станет ясно в дальнейшем, этот довольно сложный метод требуется крайне редко. Поэтому переопределять его не стоит, если только для этого нет веских оснований. В частности, метод `clone()` переопределяется лишь в менее чем 5% всех классов из стандартной библиотеки Java.

Назначение метода `clone()` — создать клон объекта, т.е. отдельный объект с таким же самым состоянием, как и у исходного объекта. Если изменить один из этих объектов, то другой останется без изменения, как показано ниже.

```
Employee cloneOfFred = fred.clone();
cloneOfFred.raiseSalary(10); // объект fred не изменился
```

Метод `clone()` объявляется в классе `Object` как `protected`. Поэтому его придется переопределить в своем классе, если пользователям класса потребуется предоставить возможность клонировать его экземпляры.

Метод `Object.clone()` создает *неполную копию* объекта. Он просто копирует все переменные экземпляра из исходного в клонируемый объект. Этого оказывается достаточно, если переменные относятся к примитивным типам или не изменяются. Но если они таковыми не являются, то оригинал и клон разделяют общее изменяемое состояние, что может вызвать осложнения. В качестве примера рассмотрим следующий класс для пересылки сообщений по электронной почте согласно заданному списку получателей:

```
public final class Message {
  private String sender;
  private ArrayList<String> recipients;
  private String text;
  ...
  public void addRecipient(String recipient) { ... };
}
```

Если сделать неполную копию объекта типа `Message`, оригинал и клон будут разделять общий список получателей сообщений (`recipients`), как показано ниже и на рис. 4.1.

```
Message specialOffer = ...;
Message cloneOfSpecialOffer = specialOffer.clone();
```

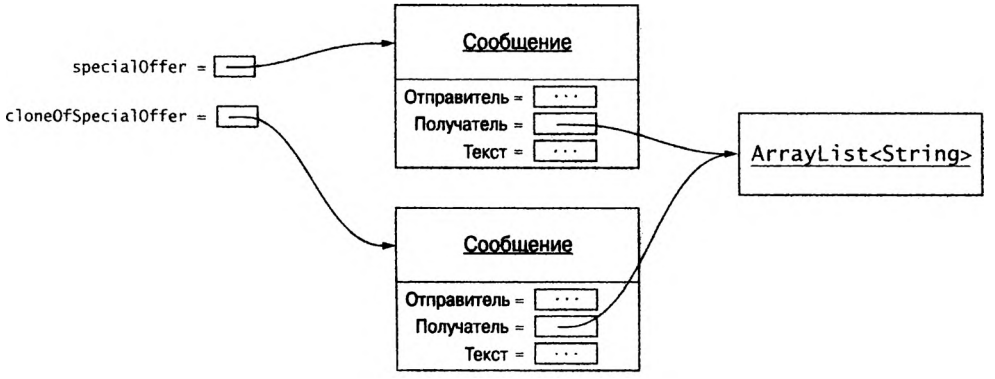


Рис. 4.1. Неполная копия объекта

Если любой из этих объектов внесет изменения в общий список получателей сообщений, эти изменения отразятся на состоянии другого объекта. Следовательно, в классе `Message` придется переопределить метод `clone()`, чтобы создать *полную копию* объекта. Но вполне возможно, что клонирование не удастся или его вообще не стоит делать. Например, клонировать объект типа `Scanner` было бы очень трудно.

В общем, реализуя класс, нужно решить, стоит ли:

1. предоставлять свой вариант метода `clone()`, или
2. удовлетвориться наследуемым его вариантом, или же
3. сделать полную копию объекта в методе `clone()`.

В первом случае вам не нужно ничего делать. Метод `clone()` наследуется вашим классом от его родительского класса, но пользователь вашего класса не сможет вызвать его, поскольку он объявлен как `protected`.

Во втором случае вам придется реализовать в своем классе интерфейс `Cloneable`. У этого интерфейса вообще отсутствуют методы, и поэтому он называется *отмечающим* или *маркерным*. (Напомним, что метод `clone()` определен в классе `Object`.) В методе `Object.clone()` проверяется, реализован ли данный интерфейс, прежде чем создавать неполную копию объекта, а иначе генерируется исключение типа `CloneNotSupportedException`. Кроме того, придется расширить область действия переопределяемого метода `clone()`, объявив его как `public` вместо `protected`, а также изменить возвращаемый им тип.

И наконец, придется каким-то образом обработать исключение типа `CloneNotSupportedException`. Это так называемое *проверяемое* исключение, и поэтому его нужно объявить или перехватить, как поясняется в главе 5. Если ваш класс объявлен как `final`, перехватить это исключение не удастся. В противном случае это исключение

можно объявить, поскольку в подклассе допускается его повторное генерирование. Ниже показано, как это делается на примере класса Employee. Приведение к типу Employee требуется потому, что метод Object.clone() возвращает объект типа Object.

```
public class Employee implements Cloneable {
  ...
  public Employee clone() throws CloneNotSupportedException {
    return (Employee) super.clone();
  }
}
```

А теперь рассмотрим третий, самый трудный случай, когда в классе нужно создать полную копию объекта. Для этого вообще не нужно пользоваться методом Object.clone(). Ниже приведен простой пример реализации метода Message.clone(). С другой стороны, можно вызвать метод clone() из суперкласса для изменяемых переменных экземпляра.

```
public Message clone() {
  Message cloned = new Message(sender, text);
  cloned.recipients = new ArrayList<>(recipients);
  return cloned;
}
```

В классе ArrayList реализуется метод clone(), возвращающий неполную копию объекта (в данном случае — списочного массива). Это означает, что исходный и клонированный списки разделяют общие ссылки на элементы списочного массива. В данном случае это приемлемо, поскольку элементы списочного массива содержат символьные строки. В противном случае придется клонировать каждый элемент объекта в отдельности, чтобы получить его полную копию.

Исторически сложилось так, что метод ArrayList.clone() возвращает объект типа Object. Поэтому его тип придется привести к нужному типу следующим образом:

```
cloned.recipients = (ArrayList<String>) recipients.clone();
    // Предупреждение о приведении типов!
```

К несчастью, такое приведение типов нельзя полностью проверить во время выполнения, как поясняется в главе 6, и поэтому будет выдано соответствующее предупреждение. Впрочем, это предупреждение можно подавить с помощью аннотации, хотя эту аннотацию можно присоединить только к объявлению (подробнее об этом — в главе 12). Ниже приведена полная реализация метода Message.clone(). В данном случае исключение типа CloneNotSupportedException не возникает, поскольку класс Message реализует интерфейс Cloneable и объявлен как final, а метод ArrayList.clone() не генерирует это исключение.

```
public Message clone() {
  try {
    Message cloned = (Message) super.clone();
    @SuppressWarnings("unchecked") ArrayList<String> clonedRecipients
         = (ArrayList<String>) recipients.clone();
```

```
    cloned.recipients = clonedRecipients;
    return cloned;
  } catch (CloneNotSupportedException ex) {
    return null; // Это исключение не может возникнуть!
  }
}
```

НА ЗАМЕТКУ. У массивов имеется открытый метод **clone()**, возвращающий такой же тип, как и у самого массива. Так, если бы список **recipients** был обычным, а не списочным массивом, его можно было бы клонировать следующим образом:

```
cloned.recipients = recipients.clone(); // Приведение типов не требуется
```

4.3. Перечисления

В главе 1 было показано, как определяются перечислимые типы. Ниже приведен типичный пример, в котором определяется тип с четырьмя конкретными экземплярами. В последующих разделах будет показано, как обращаться с перечислениями.

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

4.3.1. Методы перечислений

У каждого перечислимого типа имеется свой фиксированный ряд экземпляров, и поэтому для сравнения на равенство значений перечислимых типов метод `equals()` вообще не требуется. Для этого достаточно воспользоваться операцией ==. (При желании можно, конечно, вызвать метод `equals()`, где выполняется проверка с помощью операции ==.)

Для перечислений не нужно также предоставлять метод `toString()`. Он предоставляется автоматически для получения имени объекта перечислимого типа (в приведенном выше примере — значения `"SMALL"`, `"MEDIUM"` и т.д.)

Метод `valueOf()`, специально синтезированный для перечислений, выполняет функцию, обратную методу `toString()`. Например, в следующей строке кода:

```
Size notMySize = Size.valueOf("SMALL");
```

в переменной `notMySize` устанавливается значение перечислимого типа `Size.SMALL`. Метод `valueOf()` генерирует исключение, если отсутствует экземпляр перечислимого типа с заданным именем.

У каждого перечислимого типа имеется статический метод `values()`, возвращающий массив всех экземпляров перечисления в том порядке, в каком они были объявлены. Так, в результате следующего вызова:

```
Size[] allValues = Size.values();
```

возвращается массив с элементами `Size.SMALL`, `Size.MEDIUM` и т.д.

НА ЗАМЕТКУ. Этим методом рекомендуется пользоваться для обхода всех экземпляров перечислимого типа в расширенном цикле **for**, как показано ниже.

```
for (Size s : Size.values()) { System.out.println(s); }
```

Метод `ordinal()` возвращает позицию экземпляра в объявлении перечислимого типа `enum`, начиная с нуля. Например, в результате вызова `Size.MEDIUM.ordinal()` возвращается позиция `1`. Каждый класс перечислимого типа `E` автоматически реализует интерфейс `Comparable<E>`, допуская сравнение только с собственными объектами. Сравнение осуществляется по порядковым значениям.

НА ЗАМЕТКУ. Формально класс перечислимого типа `E` расширяет класс **`Enum<E>`**, от которого он наследует метод **`compareTo()`**, а также другие методы, описываемые в этом разделе. Методы из класса **`Enum<E>`** перечислены в табл. 4.2.

Таблица 4.2. Методы из класса **`java.lang.Enum<E>`**

Метод	Описание
`String toString()`	Имя данного экземпляра, указанное в объявлении перечислимого типа **`enum`**
`String name()`	Этот метод является конечным (**`final`**)
`int ordinal()`	Позиция данного экземпляра в объявлении перечислимого типа **`enum`**
`int compareTo(Enum<E> other)`	Сравнивает данный экземпляр с объектом ***`other`*** по порядковому значению
`static T valueOf(Class<T> type, String name)`	Возвращает экземпляр по заданному имени ***`name`***. Вместо этого лучше воспользоваться синтезированным методом **`valueOf()`** или **`values()`** перечислимого типа
`Class<E> getDeclaringClass()`	Получает класс, в котором был определен данный экземпляр. (Отличается от метода **`getClass()`**, если у экземпляра имеется тело.)
`int hashCode()`	Эти методы вызывают соответствующие методы из класса **`Object`**
`protected void finalize()`	Они являются конечными (**`final`**)
`protected Object clone()`	Генерирует исключение типа **`CloneNotSupportedException`**

4.3.2. Конструкторы, методы и поля

При желании можно ввести конструкторы, методы и поля в класс перечислимого типа, как показано в приведенном ниже примере. При этом каждый экземпляр перечисления гарантированно строится лишь один раз.

```
public enum Size {
  SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

  private String abbreviation;

  Size(String abbreviation) {
    this.abbreviation = abbreviation;
```

```
  }

  public String getAbbreviation() { return abbreviation; }
}
```

НА ЗАМЕТКУ. Конструктор перечисления всегда является закрытым. Поэтому модификатор доступа **private** в его объявлении можно опустить, как в приведенном выше примере. Объявлять конструктор перечисления как **public** или **protected** считается синтаксической ошибкой.

4.3.3. Тела экземпляров

В каждый экземпляр перечислимого типа `enum` можно ввести методы. Но они должны переопределять методы, определенные в перечислении. Например, чтобы реализовать калькулятор, можно сделать следующее:

```
public enum Operation {
  ADD {
    public int eval(int arg1, int arg2) { return arg1 + arg2; }
  },
  SUBTRACT {
    public int eval(int arg1, int arg2) { return arg1 - arg2; }
  },
  MULTIPLY {
   public int eval(int arg1, int arg2) { return arg1 * arg2; }
  },
  DIVIDE {
    public int eval(int arg1, int arg2) { return arg1 / arg2; }
  };

    public abstract int eval(int arg1, int arg2);
}
```

В цикле работы программы калькулятора одно из приведенных выше значений перечислимого типа присваивается переменной в зависимости от того, что введет пользователь, а затем вызывается метод `eval()`:

```
Operation op = ...;
int result = op.eval(first, second);
```

НА ЗАМЕТКУ. Формально каждая из перечисленных выше констант относится к анонимному подклассу, производному от класса **Operation**. Все, что можно разместить в теле анонимного подкласса, можно также ввести в тело его члена.

4.3.4. Статические члены

В перечислении допускаются статические члены. Но при этом нужно аккуратно соблюдать порядок их построения. Константы перечислимого типа строятся *перед* статическими членами, а следовательно, обратиться к любым статическим

членам в конструкторе перечисления нельзя. Например, следующий фрагмент кода недопустим:

```
public enum Modifier {
  PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL, ABSTRACT;
  private static int maskBit = 1;
  private int mask;
  public Modifier() {
    mask = maskBit; // ОШИБКА: статическая переменная
                    // в конструкторе недоступна!
    maskBit *= 2;
  }
  ...
}
```

Выходом из этого положения служит инициализация членов в статическом инициализаторе, как показано ниже. Как только константы будут построены, инициализация статических переменных и статические инициализаторы выполняются в обычном порядке сверху вниз.

```
public enum Modifier {
  PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL, ABSTRACT;
  private int mask;

  static {
    int maskBit = 1;
    for (Modifier m : Modifier.values()) {
      m.mask = maskBit;
      maskBit *= 2;
    }
  }
  ...
}
```

НА ЗАМЕТКУ. Перечислимые типы могут быть вложены в классы. В этом случае вложенные перечисления неявно считаются статическими вложенными классами, т.е. их методы не могут обращаться к переменным экземпляра объемлющего класса.

4.3.5. Переход по перечислению

Константы перечислимого типа можно использовать в операторе `switch`. В этом случае константа обозначается, например, как `ADD`, а не как `Operation.ADD` в ветви оператора `switch`, а ее тип автоматически выводится из типа выражения, по которому вычисляется переход, как показано ниже.

```
enum Operation { ADD, SUBTRACT, MULTIPLY, DIVIDE };

public static int eval(Operation op, int arg1, int arg2) {
  int result = 0;
  switch (op) {
    case ADD: result = arg1 + arg2; break;
    case SUBTRACT: result = arg1 - arg2; break;
    case MULTIPLY: result = arg1 * arg2; break;
```

```
    case DIVIDE: result = arg1 / arg2; break;
  }
  return result;
}
```

НА ЗАМЕТКУ. Согласно спецификации на язык Java, его компиляторы побуждаются к выдаче предупреждения, если переход по перечислению не является исчерпывающим, т.е. если отсутствуют ветви для всех констант и оператор **default**. Но компилятор от компании Oracle такого предупреждения не выдает.

СОВЕТ. Если к экземплярам перечисления требуется обратиться по их простому имени за пределами оператора **switch**, то для этого следует воспользоваться объявлением статического импорта. Например, с помощью следующего объявления:

```
import static com.horstmann.corejava.Size.*;
```

можно воспользоваться простым именем **SMALL** вместо **Size.SMALL**.

4.4. Динамическая идентификация типов: сведения и ресурсы

В языке Java имеется возможность выяснить во время выполнения, к какому именно классу относится заданный объект. Иногда это оказывается удобно, например, при реализации методов `equals()` и `toString()`. Более того, можно сначала выяснить, каким образом класс был загружен, а затем загрузить связанные с ним данные, называемые *ресурсами*.

4.4.1. Класс `Class`

Допустим, что имеется переменная типа `Object`, содержащая ссылку на объект, и об этом объекте требуется узнать больше, в частности, к какому классу он относится. Так, в следующем примере кода метод `getClass()` возвращает объект класса, как ни странно, называемого `Class`:

```
Object obj = ...;
Class<?> cl = obj.getClass();
```

НА ЗАМЕТКУ. Назначение суффикса **<?>** поясняется в главе 6, а до тех пор не стоит обращать на него внимание. Но опускать его нельзя, иначе применяемая ИСР выдаст неприятное предупреждение и отключит полезные проверки типов в тех выражениях, где указана переменная **cl**.

Получив в свое распоряжение объект типа `Class`, можно выяснить имя класса следующим образом:

```
System.out.println("This object is an instance of " + cl.getName());
```

С другой стороны, объект типа `Class` можно получить, вызвав статический метод `Class.forName()` таким образом:

```
String className = "java.util.Scanner";
Class<?> cl = Class.forName(className);
  // Объект, описывающий класс java.util.Scanner
```

НА ЗАМЕТКУ. Метод `Class.forName()`, как и многие другие методы, применяемые для рефлексии, генерирует проверяемые исключения, если что-нибудь пойдет не так (например, в отсутствие класса с заданным именем). В главе 5 будет разъяснено, каким образом обрабатываются исключения, а до тех пор в объявлении вызывающего метода достаточно указать следующее: `throws ReflectiveOperationException`.

Метод `Class.forName()` служит для построения объектов типа `Class`, предназначенных для получения сведений о тех классах, которые могут быть неизвестны во время компиляции. Если же заранее известно, какой именно класс требуется, то вместо данного метода можно воспользоваться *литералом класса* следующим образом:

```
Class<?> cl = java.util.Scanner.class;
```

Суффикс `.class` можно использовать для получения сведений и о других типах, как показано ниже.

```
Class<?> cl2 = String[].class; // Описывает массив типа String[]
Class<?> cl3 = Runnable.class; // Описывает интерфейс Runnable
Class<?> cl4 = int.class;      // Описывает тип int
Class<?> cl5 = void.class;     // Описывает тип void
```

В языке Java классами являются массивы, но не интерфейсы, примитивные типы и тип `void`. Имя `Class` выбрано для одноименного класса не совсем удачно, ему больше подходит имя `Type`.

ВНИМАНИЕ. Метод `getName()` возвращает не совсем обычные имена для типов массивов. Так, в результате вызова

- `String[].class.getName()` возвращается имя `"[Ljava.lang.String;"`;
- `int[].class.getName()` возвращается имя `"[I"`.

Такое обозначение употребляется в виртуальной машине с давних времен. Поэтому для получения имен вроде `"java.lang.String[]"` или `"int[]"` рекомендуется вызвать метод `getCanonicalName()`. Подобное архаичное обозначение класса употребляется при вызове метода `Class.forName()`, если для массивов требуется сформировать объекты типа `Class`.

Виртуальная машина манипулирует однозначным объектом типа `Class` для каждого типа данных. Следовательно, для сравнения объектов типа `Class`, называемых иначе *объектами класса*, можно воспользоваться операцией `==`, как показано ниже. Такое применение объектов класса уже демонстрировалось в разделе 4.2.2.

```
if (other.getClass() == Employee.class) . . .
```

В последующих разделах будет показано, что можно сделать с объектами типа `Class`. В табл. 4.3 перечислены наиболее употребительные методы из класса `Class`.

Таблица 4.3. Наиболее употребительные методы из класса `java.lang.Class<T>`

Метод	Описание
`static Class<?> forName(String className)`	Получает объект типа `Class`, описывающий класс по его имени *className*
`String getCanonicalName()`	Получают имя данного класса
`String getSimpleName()`	с дополнительными сведениями для массивов
`String getTypeName()`	внутренних классов,
`String getName()`	обобщенных классов
`String toString()`	и модификаторов доступа
`String toGenericString()`	(см. упражнение 8 в конце этой главы)
`Class<? super T> getSuperclass()`	Получают сведения о суперклассе,
`Class<?>[] getInterfaces()`	интерфейсах, пакете
`Package getPackage()`	и модификаторах доступа данного класса
`int getModifiers()`	Методы анализа значений, возвращаемых методом `getModifiers()`, перечислены в табл. 4.4
`boolean isPrimitive()`	Проверяют, является ли представленный тип примитивным или типом `void`,
`boolean isArray()`	массивом,
`boolean isEnum()`	перечислением,
`boolean isAnnotation()`	аннотацией (см. главу 12)
`boolean isMemberClass()`	классом, вложенным в другой класс,
`boolean isLocalClass()`	локальным методом или конструктором,
`boolean isAnonymousClass()`	анонимным классом
`boolean isSynthetic()`	или синтетическим классом (см. раздел 4.5.7)
`Class<?> getComponentType()`	Получают тип элемента массива,
`Class<?> getDeclaringClass()`	класса, в котором объявляется вложенный класс,
`Class<?> getEnclosingClass()`	класса, конструктора или метода,
`Constructor getEnclosingConstructor()`	в котором объявляется локальный класс
`Method getEnclosingMethod()`	
`boolean isAssignableFrom(Class<?> cls)`	Проверяют, является ли тип *cls* или класс объекта *obj*
`boolean isInstance(Object obj)`	подтипом данного типа
`T newInstance()`	Получает экземпляр данного класса, построенный конструктором без аргументов
`ClassLoader getClassLoader()`	Получает загрузчик класса, загружаемый данным классом (см. далее раздел 4.3.3)
`InputStream getResourceAsStream(String path)`	Загружают запрашиваемый ресурс из того же самого места,
`URL getResource(String path)`	откуда был загружен данный класс

Окончание табл. 4.3

Метод	Описание
`Field[] getFields()`	Получают сведения обо всех полях,
`Method[] getMethods()`	методах или же указанном поле или методе
`Field getField(String name)`	из данного класса или суперкласса
`Method getMethod(String name, Class<?>... parameterTypes)`	
`Constructor[] getConstructors()`	Получают сведения обо всех открытых конструкторах,
`Constructor[] getDeclaredConstructors()`	всех конструкторах,
`Constructor getConstructor (Class<?>... parameterTypes)`	указанном открытом конструкторе
`Constructor getDeclaredConstructor (Class<?>... parameterTypes)`	или указанном конкретном конструкторе для данного класса

Таблица 4.4. Наиболее употребительные методы из класса `java.lang.reflect.Modifier`

Метод	Описание
`static String toString(int modifiers)`	Возвращает символьную строку с модификаторами доступа, которые соответствуют отдельным двоичным разрядам, установленным в аргументе *modifiers*
`static boolean is(Abstract|Interface|Native|Private|Protected|Public|Static|Strict|Synchronized|Volatile) (int modifiers)`	Проверяет двоичный разряд, установленный в аргументе *modifiers* на позиции, соответствующей модификатору доступа в объявлении метода

4.4.2. Загрузка ресурсов

Одним из полезных применений класса `Class` является обнаружение ресурсов, которые могут понадобиться в программе, в том числе файлов конфигурации или изображений. Если разместить ресурс в том же самом каталоге, где и файл класса, то для доступа к файлу этого ресурса достаточно открыть поток ввода, как показано в следующем примере кода:

```
InputStream stream = MyClass.class.getResourceAsStream("config.txt");
Scanner in = new Scanner(stream);
```

НА ЗАМЕТКУ. Некоторые устаревшие языковые средства вроде метода **`Applet.getAudioClip()`** и конструктора класса **`javax.swing.ImageIcon`** читают данные из объекта типа **`URL`**. В таком случае можно воспользоваться методом **`getResource()`**, возвращающим URL нужного ресурса.

У ресурсов могут быть подкаталоги, доступные как по относительным, так и по абсолютным путям. Например, в результате вызова `MyClass.class.getResource AsStream("/config/menus.txt")` осуществляется поиск файла `config/menus.txt` в корневом каталоге пакета, к которому относится класс `MyClass`. Если же классы

упаковываются в архивные JAR-файлы, то нужные ресурсы следует заархивировать вместе с файлами классов, чтобы обнаружить и те и другие.

4.4.3. Загрузчики классов

Инструкции виртуальной машины хранятся в файлах классов. Каждый файл класса содержит инструкции для загрузки одного класса или интерфейса. Файл класса может находиться в файловой системе, архивном JAR-файле, удаленном месте или даже динамически формироваться в оперативной памяти. *Загрузчик классов* отвечает за загрузку отдельных байтов и их преобразования в класс или интерфейс в виртуальной машине.

Виртуальная машина загружает файлы классов по требованию, начиная с того класса, метод `main()` которого вызывается. Этот класс будет зависеть от других классов. Например, классы `java.lang.System` и `java.util.Scanner` загружаются вместе с теми классами, от которых они зависят.

В выполнении программы на Java задействованы по крайней мере три загрузчика классов. В частности, *загрузчик базовых классов* загружает классы из библиотеки (как правило, из архивного JAR-файла `jre/lib/rt.jar`) и входит в состав виртуальной машины. *Загрузчик расширений классов* загружает так называемые "стандартные расширения" из каталога `jre/lib/ext`. А *загрузчик системных классов* загружает классы из каталогов и архивных JAR-файлов по заданному пути к классу.

Для загрузчика базовых классов отсутствует соответствующий объект типа `ClassLoader`. Например, в результате вызова `String.class.getClassLoader()` возвращается пустое значение `null`. В реализации языка Java от компании Oracle загрузчики расширений классов и системных классов реализованы на Java. Оба эти загрузчика являются экземплярами класса `URLClassLoader`.

НА ЗАМЕТКУ. Помимо упомянутых выше мест, классы могут загружаться из каталога **`jre/lib/endorsed`**. Этот механизм может применяться только для замены определенных стандартных библиотек (например, библиотек для поддержки XML и CORBA) более новыми их версиями. Подробнее об этом можно узнать по следующему адресу: **`http://docs.oracle.com/javase/8/docs/technotes/guides/standards`**.

СОВЕТ. Иногда в программе требуется знать путь к классу, например, для загрузки других файлов, размещаемых относительно него. В результате вызова

```
((URLClassLoader) MainClass.class.getClassLoader()).getURLs()
```

возвращается массив объектов типа **`URL`** с каталогами и архивными JAR-файлами, доступными по пути к указанному классу.

Классы можно загрузить из каталога или архивного JAR-файла, отсутствующего в пути к классу, создав свой экземпляр класса `URLClassLoader`, как показано ниже. Это зачастую делается для загрузки подключаемых модулей.

```
URL[] urls = {
  new URL("file:///path/to/directory/"),
  new URL("file://path/to/jarfile.jar")
};
```

```
String className = "com.mycompany.plugins.Entry";
try (URLClassLoader loader = new URLClassLoader(urls)) {
  Class<?> cl = Class.forName(className, true, loader);
  // А теперь получить экземпляр объекта cl — см. раздел 4.5.4
  ...
}
```

ВНИМАНИЕ. Второй параметр в вызове **`Class.forName(className, true, loader)`** обеспечивает статическую инициализацию класса после его загрузки, что определенно требуется. Не пользуйтесь для этого методом **`ClassLoader.loadClass()`**, поскольку он *не* обеспечивает статическую инициализацию.

НА ЗАМЕТКУ. Класс **`URLClassLoader`** загружает классы из файловой системы. Если же класс требуется загрузить из какого-нибудь другого места, для этого придется написать собственный класс. В этом классе нужно реализовать лишь метод **`findClass()`**, как показано ниже.

```
public class MyClassLoader extends ClassLoader {
...
  @Override public Class<?> findClass(String name)
            throws ClassNotFoundException {
    byte[] bytes = байты из файла класса
    return defineClass(name, bytes, 0, bytes.length);
  }
}
```

В главе 14 приведен пример, в котором классы сначала компилируются в оперативной памяти, а затем загружаются.

4.4.4. Загрузчик контекста классов

Чаще всего программирующим на Java не нужно беспокоиться о процессе загрузки классов. Одни классы прозрачно загружаются по мере их надобности в других классах. Но если метод загружает классы динамически и вызывается из класса, который сам был загружен другим классом, в таком случае возможны некоторые осложнения. Обратимся к конкретному примеру.

1. Допустим, что требуется предоставить служебный класс, загружаемый загрузчиком системных классов и имеющий следующий метод:

```
public class Util {
  Object createInstance(String className) {
    Class<?> cl = Class.forName(className);
    ...
  }
  ...
}
```

2. Другой загрузчик классов загружает подключаемый модуль, читая классы из архивного JAR-файла этого модуля.

3. Подключаемый модуль вызывает метод `Util.createInstance("com.mycompany.plugins.MyClass")`, чтобы получить экземпляр класса из архивного JAR-файла этого модуля.

Автор подключаемого модуля предполагает, что класс загружен. Но в методе `Util.createInstance()` применяется свой загрузчик классов для выполнения метода `Class.forName()`, и этот загрузчик классов не будет обращаться к архивному JAR-файлу подключаемого модуля. Подобное явление называется *инверсией загрузчиков классов*.

В качестве выхода из этого положения загрузчик классов можно передать сначала служебному методу, а затем методу `forName()`:

```
public class Util {
  public Object createInstance(String className, ClassLoader loader) {
    Class<?> cl = Class.forName(className, true, loader);
    ...
  }
  ...
}
```

С другой стороны, можно воспользоваться *загрузчиком контекста классов* из текущего потока исполнения. В качестве загрузчика контекста классов в основном потоке исполнения устанавливается загрузчик системных классов. А при создании нового потока исполнения в качестве его загрузчика контекста классов устанавливается загрузчик контекста классов из создающего потока исполнения. Так, если ничего не предпринимать, то для всех потоков в качестве их загрузчика контекста классов будет служить загрузчик системных классов. Впрочем, ничто не мешает установить любой загрузчик классов, сделав следующий вызов:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

В таком случае служебный метод может извлечь загрузчик контекста классов следующим образом:

```
public class Util {
  public Object createInstance(String className) {
    Thread t = Thread.currentThread();
    ClassLoader loader = t.getContextClassLoader();
    Class<?> cl = Class.forName(className, true, loader);
    ...
  }
  ...
}
```

При вызове метода из класса подключаемого модуля в прикладной программе следует установить загрузчик классов подключаемого модуля в качестве загрузчика контекста классов, а впоследствии восстановить предыдущую установку.

СОВЕТ. Если написать метод, загружающий класс по имени, то можно просто не пользоваться загрузчиком класса этого метода. А вызывающему коду рекомендуется предоставить возможность выбора между явной передачей конкретного загрузчика классов и применением загрузчика контекста классов.

4.4.5. Загрузчики служб

Когда в прикладной программе предоставляется подключаемый модуль, разработчику подключаемого модуля нередко требуется дать свободу выбора в реализации средств этого модуля. Было бы также желательно иметь возможность выбирать среди нескольких реализаций подключаемого модуля. Класс `ServiceLoader` упрощает загрузку подключаемых модулей, соответствующих общему интерфейсу.

С этой целью следует определить интерфейс (или суперкласс, если он предпочтительнее) с методами, которые должен предоставлять каждый экземпляр службы. Допустим, что служба обеспечивает шифрование данных, как показано ниже.

```
package com.corejava.crypt;

public interface Cipher {
   byte[] encrypt(byte[] source, byte[] key);
   byte[] decrypt(byte[] source, byte[] key);
   int strength();
}
```

Поставщик служб предоставляет один или больше классов, реализующих данную службу, например, следующим образом:

```
package com.corejava.crypt.impl;

public class CaesarCipher implements Cipher {
  public byte[] encrypt(byte[] source, byte[] key) {
    byte[] result = new byte[source.length];
    for (int i = 0; i < source.length; i++)
      result[i] = (byte)(source[i] + key[0]);
    return result;
  }
  public byte[] decrypt(byte[] source, byte[] key) {
    return encrypt(source, new byte[] { (byte) -key[0] });
  }
  public int strength() { return 1; }
}
```

Классы, реализующие службы, могут находиться в любом пакете, а не обязательно в том же самом пакете, где и интерфейс службы. У каждого из них должен быть конструктор без аргументов.

А теперь следует ввести имена классов в файл, содержащий текст в кодировке UTF-8 и расположенный в каталоге `INF/services`, где его может обнаружить загрузчик классов. В данном примере файл `META-INF/services/com.corejava.crypt.Cipher` будет содержать следующую строку:

```
com.corejava.crypt.impl.CaesarCipher
```

После этой подготовки загрузчик служб инициализируется в прикладной программе, как показано ниже. Это должно быть сделано в программе лишь один раз.

```
public static ServiceLoader<Cipher> cipherLoader =
              ServiceLoader.load(Cipher.class);
```

Метод iterator() загрузчика служб предоставляет итератор для перебора всех имеющихся реализаций данной службы. (Подробнее об итераторах речь пойдет в главе 7.) Но для их перебора проще воспользоваться расширенным циклом for, где выбирается соответствующий объект для выполнения службы, как показано ниже.

```
public static Cipher getCipher(int minStrength) {
  for (Cipher cipher : cipherLoader) // неявно вызывает итератор
    if (cipher.strength() >= minStrength) return cipher;
  return null;
}
```

4.5. Рефлексия

С помощью рефлексии можно исследовать содержимое произвольных объектов во время выполнения прикладной программы и вызывать для них произвольные методы. Такая возможность удобна для реализации инструментальных средств вроде объектно-реаляционных преобразователей или построителей ГПИ. Но поскольку рефлексия интересует в основном разработчиков инструментальных средств, то прикладные программисты могут благополучно пропустить этот раздел, вернувшись к нему, когда в этом возникнет потребность.

4.5.1. Перечисление членов класса

В трех классах, Field, Method, и Constructor, из пакета java.lang.reflect описываются поля, методы и конструкторы анализируемого класса. У всех этих классов имеется метод getName(), возвращающий имя члена анализируемого класса. В классе Field имеется метод getType(), возвращающий объект, относящийся все к тому же типу Class и описывающий тип поля, а в классах Method и Constructor — методы для уведомления о типах параметров. Кроме того, класс Method способен извещать о возвращаемом типе.

У каждого из трех упомянутых выше классов имеется также метод getModifiers(), возвращающий целочисленное значение с различными установленными и сброшенными двоичными разрядами, описывающими применяемые модификаторы доступа (например, public или static). А статические методы вроде Modifier.isPublic() и Modifier.isStatic() могут быть использованы для анализа целочисленного значения, возвращаемого методом getModifiers(). И наконец, метод Modifier.toString() возвращает символьную строку со всеми применяемыми модификаторами доступа.

Методы из класса getFields(), getMethods() и getConstructors() возвращают массивы *открытых* полей; конструкторы, поддерживаемые исследуемым классом; а также открытые наследуемые члены. А методы getDeclaredFields(), getDeclaredMethods() и getDeclaredConstructors() возвращают массивы, состоящие из всех полей, методов и конструкторов, объявленных в исследуемом классе. К их числу относятся пакетные, закрытые и защищенные члены, но не члены суперклассов. И наконец, метод getParameters() из класса Executable(), служащего общим суперклассом для классов Method и Constructor, возвращает массив объектов типа Parameter, описывающих параметры методов.

НА ЗАМЕТКУ. Имена параметров доступны только во время выполнения, если класс скомпилирован с параметром командной строки **-parameters**.

В качестве примера ниже демонстрируется порядок вывода всех методов из класса. Приведенный ниже код примечателен тем, что он способен проанализировать любой класс, который может быть загружен виртуальной машиной, а не только те классы, которые были доступны на момент его компиляции.

```
Class<?> cl = Class.forName(className);
while (cl != null) {
  for (Method m : cl.getDeclaredMethods()) {
    System.out.println(
           Modifier.toString(m.getModifiers()) + " " +
           m.getReturnType().getCanonicalName() + " " +
           m.getName() + Arrays.toString(m.getParameters()));
  }
  cl = cl.getSuperclass();
}
```

4.5.2. Исследование объектов

Как было показано в предыдущем разделе, имеется возможность получить объекты типа `Field`, описывающие типы и имена полей исследуемого объекта. Но эти объекты способны на большее. В частности, из них можно извлечь значения полей. В качестве примера ниже показано, как перечислить содержимое всех полей объекта.

```
Object obj = ...;
for (Field f : obj.getClass().getDeclaredFields()) {
  f.setAccessible(true);
  Object value = f.get(obj);
  System.out.println(f.getName() + ":" + value);
}
```

Ключевая роль в данном примере кода принадлежит методу `get()`, читающему значение из поля. Если значение в поле относится к примитивному типу, то возвращается объект его оболочки. В этом случае можно также вызвать один из следующих методов: `getInt()`, `getDouble()` и т.д.

НА ЗАМЕТКУ. Закрытые объекты типа **Field** и **Method** нужно сделать доступными, прежде чем воспользоваться ими. По умолчанию виртуальная машина JVM выполняется без диспетчера безопасности, и поэтому метод **setAccessible()** разблокирует поле. Но диспетчер безопасности может заблокировать запрос, защитив объекты от доступа подобным образом.

Аналогичным образом можно установить значение в поле. В приведенном ниже примере кода производится прибавка к зарплате в объекте `obj`, независимо от того, к какому именно классу он принадлежит, но при условии, что у него имеется поле `salary` типа `double` или `Double`. В противном случае возникает исключение.

```
Field f = obj.getDeclaredField("salary");
f.setAccessible(true);
```

```
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

4.5.3. Вызов методов

Если объект типа `Field` может быть использован для чтения или записи данных в полях исследуемого объекта, то объект типа `Method` — для вызова заданного метода относительно исследуемого объекта, как показано ниже. Если же метод статический, то в качестве первоначального аргумента следует указать пустое значение `null`.

```
Method m = ...;
Object result = m.invoke(obj, arg1, arg2, ...);
```

Чтобы получить нужный метод, достаточно осуществить его поиск в массиве, возвращаемом методом `getMethods()` или `getDeclaredMethods()`, как было показано в разделе 4.5.1. С другой стороны, можно вызвать метод `getMethod()`, предоставив ему типы параметров. Так, в следующем примере демонстрируется, как получить метод `setName(String)` для объекта типа `Person`:

```
Person p = ...;
Method m = p.getClass().getMethod("setName", String.class);
p.invoke(obj, "********");
```

ВНИМАНИЕ. Несмотря на то что метод **`clone()`** является открытым во всех типах массивов, метод **`getMethod()`** ничего о нем не сообщает, когда вызывается для объекта типа **`Class`**, описывающего массив.

4.5.4. Построение объектов

Чтобы построить объект конструктором без аргументов, достаточно вызвать метод `newInstance()` для объекта типа `Class`, как показано ниже.

```
Class<?> cl = ...;
Object obj = cl.newInstance();
```

А для того чтобы вызвать другой конструктор, нужно сначала получить объект типа `Constructor`, а затем вызвать его метод `newInstance()`. Допустим, заранее известно, что у класса имеется открытый конструктор, параметр которого относится к типу `int`. В таком случае новый экземпляр можно построить следующим образом:

```
Constructor constr = cl.getConstructor(int.class);
Object obj = constr.newInstance(42);
```

Таблица 4.5. Наиболее употребительные методы из пакета `java.lang.reflect`

Класс	Метод	Примечания
`AccessibleObject`	**`void setAccessible(boolean flag)`**	Класс **`AccessibleObject`** служит суперклассом для классов **`Field`**, **`Method`**, и **`Constructor`**

Окончание табл. 4.5

Класс	Метод	Примечания
	`static void setAccessible(AccessibleObject[] array, boolean flag)`	Его методы устанавливают режим доступа к данному члену или указанным членам
`Field`	`String getName()`	Для каждого примитивного типа *p*
	`int getModifiers()`	имеются свои методы типа `get` и `set`
	`Object get(Object obj)`	
	`p getP(Object obj)`	
	`void (Object obj, Object newValue)`	
	`void setP(Object obj, p newValue)`	
`Method`	`Object invoke(Object obj, Object... args)`	Вызывает метод, описываемый данным объектом, передавая указанные аргументы и возвращая то значение, которое возвращает искомый метод. Аргументы и возвращаемые значения примитивных типов заключаются в соответствующие оболочки
`Constructor`	`Object newInstance(Object... args)`	Вызывает конструктор, описываемый данным объектом, передавая указанные аргументы и возвращая построенный объект
`Executable`	`String getName()`	Класс `Executable` служит суперклассом
	`int getModifiers()`	для классов `Method` и `Constructor`
	`Parameters[] getParameters()`	
`Parameter`	`boolean isNamePresent()`	Метод `getName()` получает заданное или синтезированное имя параметра,
	`String getName()`	например, `arg0`, если
	`Class<?> getType()`	имя отсутствует

В табл. 4.5 перечислены наиболее употребительные методы для обращения с объектами типа `Field`, `Method`, и `Constructor`.

4.5.5. Компоненты JavaBeans

Во многих объектно-ориентированных языках поддерживаются так называемые *свойства*, преобразующие выражение ***объект.ИмяСвойства*** в вызов метода получения или установки в зависимости от операции чтения и записи данных в свойстве. В языке Java такой синтаксис отсутствует, но имеется правило, согласно которому свойства соответствуют парам методов получения и установки. Компонент ***JavaBean*** — это класс с конструктором без аргументов, парами методов получения и установки и любыми другими методами.

Методы получения и установки должны следовать конкретному образцу, приведенному ниже. Но если опустить метод установки или получения, то можно получить свойства, доступные только для чтения или только для записи соответственно.

```
public Тип getСвойство()
public void setСвойство(Тип newValue)
```

Имя свойства указывается в *приводимой к нижнему регистру* форме суффикса после обозначения **get** или **set**. Например, пара методов `getSalary()` и `setSalary()` порождает свойство `salary`. Но если первые две буквы суффикса оказываются прописными, то он воспринимается буквально. Например, имя метода **getURL** обозначает получение свойства **URL**, доступного только для чтения.

НА ЗАМЕТКУ. Метод получения логических свойств можно обозначить как **get*Свойство*()** или **is*Свойство*()**, причем последнее обозначение предпочтительнее.

Компоненты JavaBeans происходят от построителей ГПИ, а их спецификация имеет загадочные правила, имеющие отношение к редакторам свойств, событиям изменения свойств и обнаружению специальных свойств. В настоящее время эти средства используются редко.

Для поддержки компонентов JavaBeans целесообразно пользоваться стандартными классами всякий раз, когда требуется обращаться с произвольными свойствами. Например, для отдельного класса можно получить объект типа `BeanInfo` следующим образом:

```
Class<?> cl = ...;
BeanInfo info = Introspector.getBeanInfo(cl);
PropertyDescriptor[] props = info.getPropertyDescriptors();
```

Чтобы получить имя и тип свойства, достаточно вызвать методы `getName()` и `getPropertyType()` для заданного объекта типа `PropertyDescriptor`. А методы `getReadMethod()` и `getWriteMethod()` возвращают объекты типа `Method` для метода получения и установки соответственно.

К сожалению, получить дескриптор метода по заданному имени свойства нельзя, поэтому придется перебрать весь массив дескрипторов следующим образом:

```
String propertyName = ...;
Object propertyValue = null;
for (PropertyDescriptor prop : props) {
  if (prop.getName().equals(propertyName))
    propertyValue = prop.getReadMethod().invoke(obj);
}
```

4.5.6. Обращение с массивами

Метод `isArray()` проверяет, представляет ли объект типа `Class` массив. Если это так, то метод `getComponentType()` возвращает объект типа `Class`, описывающий тип элементов массива. Для дальнейшего анализа или создания массива в динамическом режиме служит класс `Array` из пакета `java.lang.reflect`. В табл. 4.6 перечислены методы из этого класса.

Таблица 4.6. Методы из класса `java.lang.reflect.Array`

Метод	Описание
`static Object get(Object array, int index)`	Получают или устанавливают элемент массива по заданному индексу,
`static p getP(Object array, int index)`	где *p* — примитивный тип
`static void set(Object array, int index, Object newValue)`	
`static void setP(Object array, int index, p newValue)`	
`static int getLength(Object array)`	Получает длину заданного массива
`static Object newInstance(Class<?> componentType, int length)`	Возвращают новый массив с заданным типом элементов и размерами
`static Object newInstance(Class<?> componentType, int[] lengths)`	

В качестве упражнения реализуем в классе `Arrays` приведенный ниже метод `copyOf()`. Напомним, что с помощью этого метода можно наращивать заполненный массив.

```
Person[] friends = new Person[100];
...
// Массив заполнен
friends = Arrays.copyOf(friends, 2 * friends.length);
```

Как же написать такой обобщенный метод? Ниже приведена первая попытка.

```
public static Object[] badCopyOf(Object[] array, int newLength) {
    // Не очень полезно
  Object[] newArray = new Object[newLength];
  for (int i = 0; i < Math.min(array.length, newLength); i++)
    newArray[i] = array[i];
  return newArray;
}
```

Тем не менее воспользоваться получаемым в итоге массивом не так-то просто. Данный метод возвращает тип массива `Object[]`. Но массив объектов нельзя привести к массиву типа `Person[]`. Как упоминалось ранее, массив в Java запоминает тип своих элементов, т.е. тот тип, который указывается в операторе `new` при его создании. Массив типа `Person[]` допускается временно привести к массиву типа `Object[]`, а затем привести его обратно, но если массив изначально создан как `Object[]`, то его вообще нельзя привести к массиву типа `Person[]`.

Чтобы сделать новый массив такого же типа, как и исходный массив, потребуется метод `newInstance()` из класса `Array`. Этому методу нужно предоставить тип элементов и длину нового массива, как показано ниже.

```
public static Object goodCopyOf(Object array, int newLength) {
  Class<?> cl = array.getClass();
  if (!cl.isArray()) return null;
  Class<?> componentType = cl.getComponentType();
  int length = Array.getLength(array);
```

```
  Object newArray = Array.newInstance(componentType, newLength);
  for (int i = 0; i < Math.min(length, newLength); i++)
    Array.set(newArray, i, Array.get(array, i));
  return newArray;
}
```

Следует иметь в виду, что данный вариант метода `copyOf()` пригоден для наращивания массивов любого типа, а не только массивов объектов, как показано ниже.

```
int[] primes = { 2, 3, 5, 7, 11 };
primes = (int[]) goodCopyOf(primes, 10);
```

Параметр метода `goodCopyOf()` относится к типу `Object`, а не `Object[]`. И хотя массив типа `int[]` относится к типу `Object`, он не является массивом объектов.

4.5.7. Заместители

Класс `Proxy` позволяет создавать во время выполнения новые классы, реализующие отдельный интерфейс или ряд интерфейсов. Такие заместители нужны только в том случае, когда во время компиляции еще неизвестно, какие именно интерфейсы нужно реализовать. Ведь если нельзя получить экземпляр интерфейса, то и воспользоваться просто методом `newInstance()` не удастся.

У класса-заместителя имеются все методы, требующиеся для указанного интерфейса, а также все методы, определенные в классе `Object`, в том числе методы `toString()`, `equals()` и т.д. Но поскольку во время выполнения нельзя определить новый код для этих методов, то следует предоставить *обработчик вызовов* — объект класса, реализующего интерфейс `InvocationHandler`. У этого интерфейса имеется следующий единственный метод:

```
Object invoke(Object proxy, Method method, Object[] args)
```

При всяком вызове метода для объекта-заместителя вызывается метод `invoke()` из обработчика вызовов с объектом типа `Method` и параметрами исходного вызова. Далее обработчик вызовов должен выяснить, как обработать вызов. С этой целью обработчик вызовов может предпринять самые разные действия, в том числе направление вызовов на удаленные серверы или отслеживание вызовов для отладки.

Чтобы создать объект-заместитель, достаточно вызвать метод `newProxyInstance()` из класса `Proxy`. Этот метод принимает три следующих параметра:

- Загрузчик классов (см. раздел 4.4.3) или пустое значение `null`, чтобы использовать загрузчик классов по умолчанию.
- Массив объектов типа `Class`: по одному на каждый реализуемый интерфейс.
- Обработчик вызовов.

Ниже приведен пример, демонстрирующий механизм действия заместителей. В этом примере массив заполняется заместителями объектов типа `Integer`, а вызовы направляются исходным объектам после вывода трассировочных сообщений.

```
Object[] values = new Object[1000];

for (int i = 0; i < values.length; i++) {
```

```
  Object value = new Integer(i);
  values[i] = Proxy.newProxyInstance(
        null,
        value.getClass().getInterfaces(),
        // Лямбда-выражение для обработчика вызовов
        (Object proxy, Method m, Object[] margs) -> {
           System.out.println(value + "." + m.getName()
                              + Arrays.toString(margs));
           return m.invoke(value, margs);
        });
}
```

Следующий вызов:

```
Arrays.binarySearch(values, new Integer(500));
```

приводит к выводу таких результатов:

```
499.compareTo[500]
749.compareTo[500]
624.compareTo[500]
561.compareTo[500]
530.compareTo[500]
514.compareTo[500]
506.compareTo[500]
502.compareTo[500]
500.compareTo[500]
```

Как видите, алгоритм двоичного поиска сосредоточен на искомом ключе, сокращая наполовину интервал поиска на каждому шагу. Дело в том, что метод `compareTo()` вызывается через заместителя, несмотря на то, что он не указан в коде явно. Все методы из любого интерфейса, реализуемого классом `Integer`, замещаются.

ВНИМАНИЕ. Если обращение к обработчику вызовов происходит при вызове метода без параметров, то массив аргументов оказывается пустым (`null`), а не массивом типа `Object[]` нулевой длины. Этого поведения следует всячески избегать в прикладном коде как достойного порицания.

Упражнения

1. Определите класс `Point` с конструктором `Point(double x, double y)` и методами доступа `getX()`, `getY()`. Определите также подкласс `LabeledPoint` с конструктором `LabeledPoint(String label, double x, double y)` и методом доступа `getLabel()`.
2. Определите методы `toString()`, `equals()` и `hashCode()` для классов из предыдущего упражнения.
3. Объявите как `protected` переменные экземпляра `x` и `y` из класса `Point` в упражнении 1. Продемонстрируйте, что эти переменные доступны классу `LabeledPoint` только в его экземплярах.

4. Определите абстрактный класс `Shape` с переменной экземпляра класса `Point`; конструктором и конкретным методом `public void moveBy(double dx, double dy)`, перемещающим точку на заданное расстояние; а также абстрактным классом `public Point getCenter()`. Предоставьте конкретные подклассы `Circle`, `Rectangle`, `Line` с конструкторами `public Circle(Point center, double radius)`, `public Rectangle(Point topLeft, double width, double height)` и `public Line(Point from, Point to)`.
5. Определите методы `clone()` в классах из предыдущего упражнения.
6. Допустим, что в методе `Item.equals()`, представленном в разделе 4.2.2, используется проверка с помощью операции `instanceof`. Реализуйте метод `DiscountedItem.equals()` таким образом, чтобы выполнять в нем сравнение только с суперклассом, если его параметр `otherObject` относится к типу `Item`, но с учетом скидки, если это тип `DiscountedItem`. Продемонстрируйте, что этот метод сохраняет симметричность, но не *транзитивность*, т.е. способность обнаруживать сочетание товаров по обычной цене и со скидкой, чтобы делать вызовы `x.equals(y)` и `y.equals(z)`, но не `x.equals(z)`.
7. Определите перечислимый тип для восьми комбинаций основных цветов — `BLACK`, `RED`, `BLUE`, `GREEN`, `CYAN`, `MAGENTA`, `YELLOW`, `WHITE` — с методами `getRed()`, `getGreen()` и `getBlue()`.
8. В классе `Class` имеются шесть методов, возвращающих строковое представление типа, описываемого объектом типа `Class`. Чем отличается их применение к массивам, обобщенным типам, внутренним классам и примитивным типам?
9. Напишите "универсальный" метод `toString()`, в котором применяется рефлексия для получения символьной строки со всеми переменными экземпляра объекта. В качестве дополнительного задания можете организовать обработку циклических ссылок.
10. Воспользуйтесь примером кода из раздела 4.5.1 для перечисления всех методов из класса типа `int[]`. В качестве дополнительного задания можете выявить один метод, обсуждавшийся в этой главе, как неверно описанный.
11. Напишите программу, выводящую сообщение `"Hello, World"`, воспользовавшись рефлексией для поиска поля `out` в классе `java.lang.System` и методом `invoke()` для вызова метода `println()`.
12. Определите отличие в производительности обычного вызова метода от его вызова через рефлексию.
13. Напишите метод, выводящий таблицу значений из любого объекта типа `Method`, описывающего метод с параметром типа `double` или `Double`. Помимо объекта типа `Method`, этот метод должен принимать нижний и верхний предел, а также величину шага. Продемонстрируйте свой метод, выведя таблицы для методов `Math.sqrt()` и `Double.toHexString()`. Напишите еще один вариант данного метода, но на этот раз воспользуйтесь объектом типа `DoubleFunction <Object>` вместо объекта типа `Method` (см. раздел 3.6.2). Сопоставьте безопасность, эффективность и удобство обоих вариантов данного метода.

Глава 5

Исключения, утверждения и протоколирование

В этой главе...

Во многих программах приходится иметь дело с неожиданными ситуациями, которые могут оказаться намного сложнее реализации так называемого "счастливого" сценария. Как и в большинстве современных языков программирования, в Java имеется надежный механизм обработки исключений для передачи управления из точки отказа компетентному обработчику. Кроме того, в Java имеется оператор `assert`, обеспечивающий структурированный и эффективный способ выражения внутренних допущений. И наконец, в этой главе будет показано, как пользоваться прикладным программным интерфейсом API для протоколирования различных (предусмотренных или подозрительных) событий при выполнении прикладных программ.

Основные положения этой главы приведены ниже.

1. Когда генерируется исключение, управление передается ближайшему обработчику исключений.
2. В языке Java проверяемые исключения отслеживаются компилятором.
3. Для обработки исключений служит конструкция в виде блока операторов `try/catch`.
4. Оператор `try` с ресурсами автоматически закрывает ресурсы после нормального выполнения кода или при возникновении исключения.
5. Блок операторов `try/finally` служит для других действий, которые должны произойти независимо того, будет ли выполнение кода продолжено нормально.
6. Исключение можно перехватывать и генерировать повторно или связывать его в цепочку с другим исключением.
7. Трассировка стека описывает все вызовы методов, ожидающие в точке исполнения.
8. Утверждение служит для проверки условия, если только такая проверка разрешена для класса. И если условие не выполняется, то генерируется ошибка.
9. Регистраторы организованы в иерархию и могут получать протокольные сообщения с разными уровнями протоколирования в пределах от `SEVERE` до `FINEST`.
10. Обработчики протоколов могут посылать протокольные сообщения разным адресатам, а средства форматирования — определять формат этих сообщений.
11. Свойства протоколирования можно задавать в файле конфигурации.

5.1. Обработка исключений

Что должен сделать метод, если возникнет ситуация, когда он не в состоянии выполнить свой контракт? Традиционный ответ на этот вопрос раньше был следующим: метод должен возвратить код ошибки вызвавшему его коду. Но вызывать такой метод неудобно. Ведь на вызывающий код возлагается обязанность проверять ошибки, и если он не способен их обработать, то должен возвратить их тому коду, который его вызвал. Поэтому и не удивительно, что программисты не всегда проверяли и распространяли коды ошибок, в результате чего ошибки оставались необнаруженными, приводя к аварийной ситуации.

Вместо распространения кодов ошибок вверх по цепочке вызовов методов в Java поддерживается *обработка исключений*, при которой метод может известить о серьезной ошибке, сгенерировав исключение. Один из методов в цепочке вызовов, но не обязательно непосредственно вызвавший данный метод, обязан обработать исключение, перехватив его. Главное преимущество обработки исключений состоит в том, что она развязывает процессы обнаружения и обработки ошибок. В последующих разделах будет показано, как обращаться с исключениями в Java.

5.1.1. Генерирование исключений

Метод может сам выявить ситуацию, когда он не в состоянии дальше выполнять возложенную на него задачу. Возможно, для этого ему не хватает какого-нибудь ресурса или же ему были переданы несовместимые параметры. В таком случае исключение лучше сгенерировать в самом методе.

Допустим, что требуется реализовать метод, возвращающий случайное целое число в заданных пределах, как показано ниже.

```
private static Random generator = new Random();

public static int randInt(int low, int high) {
  return low + (int) (generator.nextDouble() * (high - low + 1))
}
```

Что должно произойти, если кто-нибудь сделает вызов `randInt(10, 5)`? Пытаться устранить подобную ошибку в вызывающем коде вряд ли стоит, поскольку это не единственная возможная ошибка. Вместо этого лучше сгенерировать подходящее исключение следующим образом:

```
if (low > high)
  throw new IllegalArgumentException(
     String.format("low should be <= high but low is %d and high is %d",
                   low, high));
```

Как видите, оператор `throw` служит для генерирования объекта класса исключения (в данном случае — `IllegalArgumentException`). Этот объект строится вместе с отладочным сообщением. В следующем разделе будет показано, как выбирать класс подходящего исключения.

Когда выполняется оператор `throw`, нормальный порядок исполнения кода немедленно прерывается. В частности, метод `randInt()` прекращает свое выполнение и не возвращает значение вызывающему его коду. Вместо этого управление передается обработчику исключений, как будет показано далее, в разделе 5.1.4.

5.1.2. Иерархия исключений

На рис. 5.1 демонстрируется иерархия классов исключений в Java. Все исключения являются подклассами, производными от класса `Throwable`. А подклассы, производные от класса `Error`, представляют исключения, генерируемые в тех исключительных ситуациях, справиться с которыми программа не в состоянии, например,

с исчерпанием оперативной памяти. В подобных ситуациях остается только выдать пользователю сообщение о серьезной, непоправимой ошибке.

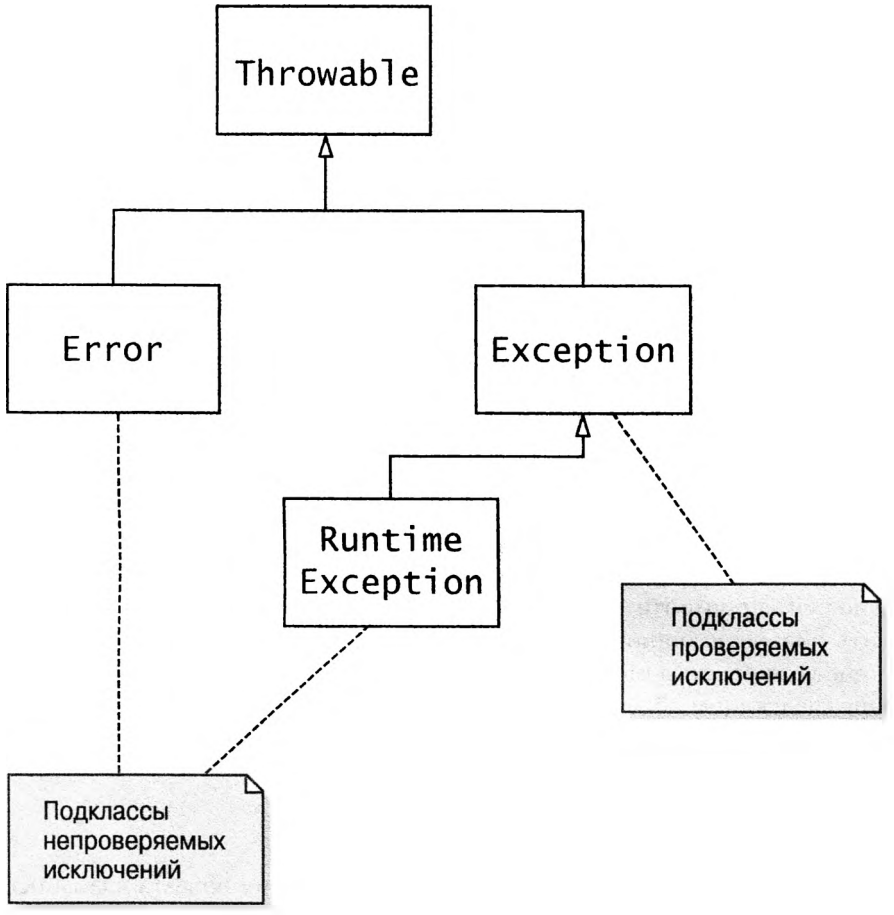


Рис. 5.1. Иерархия классов исключений

Исключения, обрабатываемые программно, представлены подклассами, производными от класса `Exception`. Такие исключения разделяются на две категории.

- *Непроверяемые* исключения, представленные подклассами, производными от класса `RuntimeException`.
- Все остальные исключения, которые считаются *проверяемыми*.

Как будет показано в следующем разделе, программисты должны перехватить проверяемые исключения или объявить их в заголовке метода. А компилятор проверит, обрабатываются ли эти исключения надлежащим образом.

НА ЗАМЕТКУ. Имя класса **`RuntimeException`** выбрано не совсем удачно. Безусловно, все исключения происходят во время выполнения. Но исключения, представленные подклассами, производными от класса **`RuntimeException`**, не проверяются во время компиляции.

Проверяемые исключения применяются в тех случаях, когда возможный отказ должен быть предвиден заранее. Одной из типичных причин такого отказа может служить ввод-вывод данных. В процессе ввода-вывода файлы могут испортиться, а сетевые соединения — пропасть. Класс `IOException` расширяет целый ряд классов исключений, и поэтому для извещения о любых обнаруженных ошибках следует выбрать наиболее подходящий класс. Например, если файл отсутствует там, где он должен быть, генерируется исключение типа `FileNotFoundException`.

Непроверяемые исключения указывают на логические ошибки, совершаемые программистами, а не в связи с неизбежными внешними рисками. Например, исключение типа `NullPointerException` является непроверяемым. Практически любой метод может сгенерировать такое исключение, и тратить время на его перехват нецелесообразно. Вместо этого лучше сначала проверить, не происходит ли разыменование пустых указателей.

Иногда разработчикам нужно провести ясное различие между проверяемыми и непроверяемыми исключениями. Рассмотрим в качестве примера вызов метода `Integer.parseInt(str)`. Этот метод генерирует непроверяемое исключение типа `NumberFormatException`, если параметр `str` содержит недостоверное целочисленное значение. С другой стороны, в результате вызова `Class.forName(str)` генерируется проверяемое исключение типа `ClassNotFoundException`, если параметр `str` содержит достоверное имя класса.

Зачем же проводить различие между этими исключениями? А затем, что проверить, содержит ли символьная строка целочисленное значение, можно перед тем, как вызывать метод `Integer.parseInt()`. Но узнать, может ли класс быть загружен, нельзя до тех пор, пока не попытаться загрузить его.

В прикладном программном интерфейсе Java API предоставляется немало классов исключений, в том числе `IOException`, `IllegalArgumentException` и т.п. Ими следует пользоваться, когда это уместно. Но если ни один из стандартных классов исключений непригоден для преследуемых целей, то можно создать свой класс, расширяющий класс `IOException`, `IllegalArgumentException` или другой существующий класс исключения.

Такой класс целесообразно снабдить конструктором по умолчанию, а также конструктором, принимающим в качестве параметра символьную строку с сообщением, как в следующем примере кода:

```
public class FileFormatException extends IOException {
  public FileFormatException() {}
  public FileFormatException(String message) {
    super(message);
  }
  // Добавить также конструкторы для цепочек исключений —
  // см. далее раздел 5.1.7
}
```

5.1.3. Объявление проверяемых исключений

Проверяемое исключение должно быть объявлено в заголовке любого метода, способного сгенерировать это исключение, с помощью оператора `throws`, как показано

ниже. В этом операторе перечисляются все исключения, которые метод может сгенерировать явно в операторе `throw` или неявно в результате вызова другого метода с оператором `throws`.

```
public void write(Object obj, String filename)
    throws IOException, ReflectiveOperationException
```

В операторе `throws` можно указывать исключения из общего суперкласса. Насколько это целесообразно делать, зависит от конкретных исключений. Так, если метод может сгенерировать несколько исключений, представленных подклассами, производными от класса `IOException`, то все эти исключения имеет смысл охватить одним оператором `throws IOException`. Но если исключения не связаны вместе, то объединить их в одном операторе `throws IOException` — означает лишить всякого смысла назначение проверки исключений.

СОВЕТ. Некоторые программисты считают зазорным признать, что метод может сгенерировать исключение. Не лучше ли его вместо этого обработать? Но самом деле верно обратное. Каждое исключение должно найти свой путь к компетентному обработчику. Золотое правило для исключений таково: генерировать раньше, перехватывать позже.

Если метод переопределяется, он не может генерировать больше проверяемых исключений, чем их объявлено в методе из суперкласса. Так, если попытаться расширить упомянутый выше метод `write()`, то в следующем переопределяющем его методе может быть сгенерировано меньше исключений:

```
public void write(Object obj, String filename)
    throws FileNotFoundException
```

Но если попытаться сгенерировать в таком методе другое исключение, не связанное с объявленным, например, исключение типа `InterruptedException`, то этот метод не будет скомпилирован.

ВНИМАНИЕ. Если в методе из суперкласса отсутствует оператор **`throws`**, то ни один из переопределяющих его методов не может сгенерировать проверяемое исключение.

Для документирования условий генерирования в методе (проверяемого или непроверяемого) исключения можно воспользоваться аннотацией `@throws`. Но большинство программирующих на Java делают это только в том случае, если в прикладном коде имеется нечто, заслуживающее документирования. Вряд ли, например, имеет смысл сообщать пользователям метода, что исключение типа `IOException` генерируется в том случае, если возникает ошибка ввода-вывода. Но комментарии вроде следующего могут вполне иметь свой смысл:

```
@throws Исключение типа NullPointerException, если имя файла пустое
@throws Исключение типа FileNotFoundException, если файл с указанным
        именем отсутствует
```

НА ЗАМЕТКУ. Тип исключения в лямбда-выражении вообще не объявляется. Но если лямбда-выражение может сгенерировать проверяемое исключение, то его можно передать только функциональному интерфейсу, в методе которого это исключение объявляется. Например, следующий вызов является ошибочным:

```
list.forEach(obj -> write(obj, "output.dat"));
```

В качестве параметра методу **forEach()** передается приведенный ниже функциональный интерфейс, а объявленный в нем метод **accept()** не генерирует никакого проверяемого исключения.

```
public interface Consumer<T> {
   void accept(T t);
}
```

5.1.4. Перехват исключений

Чтобы перехватить исключение, достаточно задать блок операторов try/catch. В простейшей форме этот блок выглядит следующим образом:

```
try {
  операторы
} catch (КлассИсключения ex) {
  обработчик исключений
}
```

Если возникает исключение заданного класса, выполняются операторы из блока try, и управление передается обработчику данного типа исключений. Переменная исключения (в данном случае ex) ссылается на объект исключения, который может проверить, если требуется, обработчик исключений.

Имеются две разновидности приведенной выше основной конструкции блока операторов try/catch. С одной стороны, можно указать несколько обработчиков разнотипных исключений следующим образом:

```
try {
  операторы
} catch (КлассИсключения1 ex) {
  обработчик исключения1
} catch (КлассИсключения2 ex) {
  обработчик исключения2
} catch (КлассИсключения3 ex) {
  обработчик исключения3
}
```

Операторы catch располагаются сверху вниз таким образом, чтобы первым было перехвачено наиболее конкретное исключение. С другой стороны, разнотипные исключения можно обрабатывать в общем обработчике приведенным ниже образом. В этом случае в обработчике исключений можно вызывать только те методы для переменной исключения, которые относятся ко всем классам исключений.

```
try {
  операторы
} catch (КлассИсключения1 | КлассИсключения2 | КлассИсключения3 ex) {
  обработчик исключений
}
```

5.1.5. Оператор try с ресурсами

Обработку исключений, в частности, затрудняет управление ресурсами. Допустим, что в файл нужно записать данные, а по завершении этой операции — закрыть файл, как показано ниже.

```
ArrayList<String> lines = ...;
PrintWriter out = new PrintWriter("output.txt");
for (String line : lines) {
  out.println(line.toLowerCase());
}
out.close();
```

В приведенном выше примере кода кроется следующая опасность: если в любом методе генерируется исключение, вызов `out.close()` вообще не происходит, а это плохо. Этот вызов может быть пропущен. И если исключение возникает неоднократно, то система может исчерпать дескрипторы файлов.

Это затруднение позволяет разрешить специальная форма оператора `try`. В заголовке такого оператора можно указать любое количество ресурсов следующим образом:

```
try (ТипРесурса1 res1 = init1; ТипРесурса2 res2 = init2; ...) {
  операторы
}
```

Каждый ресурс должен относиться к классу, реализующему интерфейс `AutoCloseable`. У этого интерфейса имеется следующий единственный метод:

```
public void close() throws Exception
```

НА ЗАМЕТКУ. Имеется также интерфейс **`Closeable`**. Этот подинтерфейс является производным от интерфейса **`AutoCloseable`** и также имеет единственный метод **`close()`**. Но этот метод объявляется для генерирования исключения типа **`IOException`**.

Если имеется блок оператора `try`, то методы `close()` вызываются для всех объектов ресурсов независимо от того, достигается ли конец блока оператора `try` нормально или же генерируется исключение. Так, в следующем примере кода блок оператора `try` гарантирует, что метод `out.close()` вызывается в любом случае:

```
ArrayList<String> lines = ...;
try (PrintWriter out = new PrintWriter("output.txt")) {
  for (String line : lines) {
    out.println(line.toLowerCase());
  }
}
```

А в приведенном ниже примере задействованы два ресурса. Эти ресурсы закрываются в обратном порядке их инициализации, т.е. метод `out.close()` вызывается перед методом `in.close()`.

```
try (Scanner in = new Scanner(Paths.get("/usr/share/dict/words"));
     PrintWriter out = new PrintWriter("output.txt")) {
  while (in.hasNext())
    out.println(in.next().toLowerCase());
}
```

Допустим, что в конструкторе класса `PrintWriter` генерируется исключение. В таком случае ресурс `in` инициализируется, а ресурс `out` не инициализируется. Тем не менее оператор `try` с ресурсами действует правильно, вызывая метод `in.close()` и распространяя исключение.

Некоторые методы `close()` способны генерировать исключения. В этом случае сгенерированное исключение передается вызывающему коду, когда блок оператора `try` завершается нормально. Но если сгенерировано другое исключение, приводящее к вызову методов `close()` используемых ресурсов, а один из этих методов генерирует исключение, то такое исключение, вероятнее всего, окажется менее важным, чем первоначальное.

В таком случае первоначальное исключение генерируется повторно, а исключения, возникающие в результате вызова методов `close()`, перехватываются и присоединяются как " подавленные". Это очень удобный механизм, реализовать который вручную было бы совсем не просто (см. упражнение 5 в конце этой главы). При перехвате первичного исключения вторичные исключения можно извлечь, вызвав метод `getSuppressed()` следующим образом:

```
try {
  ...
} catch (IOException ex) {
  Throwable[] secondaryExceptions = ex.getSuppressed();
  ...
}
```

Если подобный механизм требуется реализовать вручную в ситуации, хотя и крайне редкой, когда оператором `try` с ресурсами воспользоваться нельзя, следует сделать вызов `ex.addSuppressed(secondaryException)`. Оператор `try` с ресурсами может иметь дополнительные операторы `catch`, перехватывающие любые исключения в данном операторе.

5.1.6. Оператор `finally`

Как было показано выше, оператор `try` с ресурсами автоматически закрывает ресурсы независимо от того, возникает ли исключение или нет. Но иногда требуется очистить какой-нибудь ресурс, не закрываемый автоматически средствами интерфейса `AutoCloseable`. В таком случае можно воспользоваться оператором `finally`, как показано ниже. Оператор `finally` выполняется по завершении блока оператора `try` нормальным образом или вследствие исключения.

```
try {
  выполнить действия
} finally {
  очистить ресурс
}
```

По такому образцу следует действовать всякий раз, когда требуется установить и снять блокировку, инкрементировать и декрементировать счетчик или разместить данные в стеке и затем извлечь их из стека по завершении основной операции. Следует, однако, иметь в виду, что подобные действия происходят независимо от типа генерируемых исключений.

Следует также избегать генерирования исключения в операторе `finally`, Если блок оператора `try` завершается вследствие исключения, то последнее маскируется исключением в операторе `finally`. Механизм подавления исключений, рассмотренный в предыдущем разделе, пригоден только для операторов `try` с ресурсами.

Аналогично оператор `finally` не должен содержать оператор `return`. Если в блоке оператора `try` также имеется оператор `return`, то возвращаемое им значение заменяется в одном из операторов `finally`.

Операторы `try` можно составить в блок с операторами `catch`, а завершить их оператором `finally`. Но тогда придется очень осторожно обращаться с исключениями в операторе `finally`. В качестве примера рассмотрим следующий блок оператора `try`, адаптированный из оперативно доступного учебного материала:

```
BufferedReader in = null;
try {
  in = Files.newBufferedReader(path, StandardCharsets.UTF_8);
  Читать из стандартного потока ввода in
} catch (IOException ex) {
  System.err.println("Caught IOException: " + ex.getMessage());
} finally {
  if (in != null) {
    in.close(); // ВНИМАНИЕ: может быть сгенерировано исключение!
  }
}
```

Автор этого кода, очевидно, рассчитывал на тот случай, когда метод `Files.newBufferedReader()` сгенерирует исключение. Но если в этом коде будут перехвачены и выведены все исключения ввода-вывода, то среди них может быть пропущено исключение, генерируемое в методе `in.close()`. Поэтому сложный блок операторов `try/catch/finally` лучше переписать, превратив его в блок оператора `try` с ресурсами или вложив блок операторов `try/finally` в блок операторов `try/catch` (см. упражнение 6 в конце этой главы).

5.1.7. Повторное генерирование и связывание исключений в цепочку

Когда возникает исключение, возможно, еще неизвестно, что с ним делать, но требуется запротоколировать отказ. В таком случае следует сгенерировать исключение повторно, чтобы передать его компетентному обработчику, как показано ниже.

```
try {
  Выполнить действия
}
catch (Exception ex) {
  logger.log(level, message, ex);
  throw ex;
}
```

НА ЗАМЕТКУ. Если разместить приведенный выше код в теле метода, который может сгенерировать исключение, то произойдет нечто едва уловимое. Допустим, что объемлющий метод объявлен следующим образом:

```
public void read(String filename) throws IOException
```

На первый взгляд, достаточно заменить оператор **throws** на оператор **throws Exception**. Но компилятор Java тщательно проверяет последовательность операций и приходит к выводу, что в блоке оператора **try** может быть сгенерировано только исключение **ex**, а не произвольное исключение типа **Exception**.

Иногда требуется изменить класс генерируемого исключения. Например, нужно известить систему об отказе, используя класс исключения, которое имеет определенный смысл для пользователя подсистемы. Допустим, что в сервлете возникает ошибка обращения к базе данных. Коду, выполняющему сервлет, хорошо известна причина ошибки, но ему нужно точно знать, что сервлет находится в состоянии отказа. В таком случае следует перехватить первоначальное исключение и связать его в цепочку с исключением более высокого уровня, как показано ниже.

```
try {
  Доступ к базе данных
}
catch (SQLException ex) {
  throw new ServletException("database error", ex);
}
```

Когда перехватывается исключение типа `ServletException`, первоначальное исключение может быть извлечено следующим образом:

```
Throwable cause = ex.getCause();
```

У класса `ServletException` имеется конструктор, принимающий в качестве параметра причину, по которой возникло исключение. В данном случае придется вызвать метод `initCause()` следующим образом:

```
try {
 Доступ к базе данных
}
catch (SQULexception ex) {
  Throwable ex2 = new CruftyOldException("database error");
  ex2.initCause(ex);
  throw ex2;
}
```

Если же предоставить свой класс исключения, то, помимо двух конструкторов, описанных ранее в разделе 5.1.2, его придется снабдить следующими конструкторами:

```
public class FileFormatException extends IOException {
  ...
  public FileFormatException(Throwable cause) { initCause(cause); }
  public FileFormatException(String message, Throwable cause) {
    super(message);
    initCause(cause);
  }
}
```

СОВЕТ. Механизм связывания исключений в цепочку оказывается удобным в том случае, если проверяемое исключение происходит в методе, где запрещено генерировать проверяемое исключение. Такое исключение можно перехватить и связать его в цепочку с непроверяемым исключением.

5.1.8. Трассировка стека

Если исключение больше нигде не перехватывается, то выводится результат *трассировки стека* — перечень всех зависших вызовов методов на момент генерирования исключения. Результат трассировки стека направляется в стандартный поток вывода ошибок `System.err`.

Если исключение требуется сохранить в каком-нибудь другом месте, возможно, для дальнейшего анализа персоналом технической поддержки программного обеспечения, то для этой цели можно установить обработчик необрабатываемых исключений по умолчанию следующим образом:

```
Thread.setDefaultUncaughtExceptionHandler((thread, ex) -> {
  Зарегистрировать исключение
});
```

НА ЗАМЕТКУ. Необрабатываемое исключение прекращает исполнение того потока, в котором оно произошло. Если в прикладной программе имеется единственный поток исполнения (как в приведенных до сих пор примерах программ), то данная программа завершается после вызова обработчика необрабатываемых исключений.

Иногда приходится перехватывать исключение, даже не зная, что с ним делать. Например, метод `Class.forName()` генерирует проверяемое исключение, которое нужно каким-то образом обработать. Вместо того чтобы пренебречь этим исключением, достаточно хотя бы вывести результат трассировки стека, как показано ниже.

```
try {
  Class<?> cl = Class.forName(className);
  ...
} catch (ClassNotFoundException ex) {
  ex.printStackTrace(new PrintStream(out));
}
```

Если же результат трассировки стека данного исключения требуется сохранить, его можно вывести символьной строкой следующим образом:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
ex.printStackTrace(out);
String description = out.toString();
```

НА ЗАМЕТКУ. Если результат трассировки стека требуется обработать более тщательно, то сначала нужно сделать следующий вызов:

```
StackTraceElement[] frames = ex.getStackTrace();
```

а затем проанализировать экземпляры класса **`StackTraceElement`**. Подробнее об этом см. в документации на прикладной программный интерфейс Java API.

5.1.9. Метод `Objects.requireNonNull()`

В классе `Objects` имеется метод, удобный для проверки параметров на пустые значения. Ниже приведен характерный пример применения этого метода.

```
public void process(String directions) {
  this.directions = Objects.requireNonNull(directions);
  ...
}
```

Если переменная экземпляра `directions` содержит пустое значение `null`, то генерируется исключение типа `NullPointerException`, что, на первый взгляд, кажется не таким уж и существенным улучшением. Но допустим, что требуется вернуться к нормальной работе после трассировки стека. Если причина сбоя кроется в вызове метода `requireNonNull()`, то сразу становится ясной ошибка. Генерируемое исключение можно снабдить соответствующей строкой сообщения, как показано ниже.

```
this.directions = Objects.requireNonNull(
                  directions, "directions must not be null");
```

5.2. Утверждения

Зачастую утверждения служат явным признаком стиля защитного программирования. Допустим, что выполнение прикладного кода опирается на конкретное свойство. Например, требуется извлечь квадратный корень числа следующим образом:

```
double y = Math.sqrt(x);
```

При этом нет никаких сомнений, что значение параметра `x` не является отрицательным числом. Тем не менее нужно еще раз убедиться, что в вычисление не проникли числовые значения с плавающей точкой, не являющиеся отрицательными. С этой целью можно было бы, конечно, сгенерировать исключение следующим образом:

```
if (x < 0) throw new IllegalStateException(x + " < 0");
```

НА ЗАМЕТКУ. В языке Java утверждения служат для целей отладки и проверки достоверности внутренних утверждений, а не в качестве механизма для соблюдения контрактов. Так, если требуется сообщить о неподходящем параметре открытого метода, то вместо утверждения лучше сгенерировать исключение типа **`IllegalArgumentException`**.

Но такое положение остается в программе даже после того, как ее тестирование будет завершено, замедляя ее выполнение. Механизм утверждений позволяет вводить проверки во время тестирования и удалять их автоматически в выходном коде.

5.2.1. Применение утверждений

В языке Java предусмотрены две формы оператора утверждения:

```
assert условие;
assert условие: выражение;
```

Оператор `assert` вычисляет заданное ***условие*** и генерирует исключение типа `AssertionError`, если ***условие*** ложно. Во второй форме заданное ***выражение*** преобразуется в символьную строку сообщения об ошибке.

НА ЗАМЕТКУ. Если выражение относится к типу `Throwable`, то оно также задается как причина ошибки утверждения (см. выше раздел 5.1.7).

Например, чтобы сделать утверждение, что значение переменной `x` не является отрицательным числом, достаточно воспользоваться следующим оператором:

```
assert x >= 0;
```

С другой стороны, конкретное значение переменной `x` можно передать объекту типа `AssertionError`, чтобы отобразить его в дальнейшем, как показано ниже.

```
assert x >= 0 : x;
```

5.2.2. Разрешение и запрет утверждений

По умолчанию утверждения запрещаются. Чтобы разрешить их, достаточно запустить программу на выполнение с параметром командной строки **`-enableassertions`** (или **`-ea`**) следующим образом:

```
java -ea MainClass
```

Для этого совсем не обязательно перекомпилировать программу, поскольку разрешение или запрет утверждений обрабатывается загрузчиком классов. Если утверждения запрещены, загрузчик классов вычленяет код утверждений, чтобы он не замедлял выполнение программы. Имеется даже возможность разрешить утверждения в отдельных классах или целых пакетах, как показано в следующем примере команды:

```
java -ea:MyClass -ea:com.mycompany.mylib... MainClass
```

Эта команда разрешает утверждения для класса `MyClass` и всех классов из пакета `com.mycompany.mylib` и подчиненных ему пакетов. Параметр **`-ea`** разрешает утверждения во всех классах из пакета по умолчанию.

С помощью параметров командной строки **-disableassertions** или **-da** можно также запретить утверждения в некоторых классах и пакетах, как показано ниже.

```
java -ea:... -da:MyClass MainClass
```

Если параметры **-disableassertions** или **-da** употребляются для разрешения или запрета всех утверждений (а не только в отдельных классах или пакетах), их действие не распространяется на так называемые "системные" классы, загружаемые без помощи загрузчиков классов. Для того чтобы разрешить утверждения в системных классах, следует воспользоваться параметром командной строки **-enable systemassertions** (**-esa**). Имеется также возможность для программного управления состоянием утверждений в загрузчиках классов с помощью следующих методов:

```
void ClassLoader.setDefaultAssertionStatus(boolean enabled);
void ClassLoader.setClassAssertionStatus(
                          String className, boolean enabled);
void ClassLoader.setPackageAssertionStatus(
                            String packageName, boolean enabled);
```

Как и параметр командной строки **-enableassertions**, метод `setPackageAssertionStatus()` задает состояние утверждений для отдельного пакета и его подпакетов.

5.3. Протоколирование

Всякому программирующему на Java известен процесс ввода вызовов метода `System.out.println()` в приводящий к недоразумениям код, чтобы получить ясное представление о поведении программы. Разумеется, как только причина недоразумений будет устранена, операторы с подобными вызовами могут быть удалены из проверяемого кода, хотя их придется снова ввести в код при возникновении в нем новых затруднений. Это неудобство призван устранить прикладной программный интерфейс API для протоколирования.

5.3.1. Применение регистраторов

Начнем рассмотрение протоколирования с простейшего примера. Система протоколирования управляет регистратором по умолчанию, получаемым в результате вызова метода `Logger.getGlobal()`. А для протоколирования информационного сообщения служит метод `info()`, как показано ниже.

```
Logger.getGlobal().info("Opening file " + filename);
```

В итоге выводится запись, аналогичная следующей:

```
Aug 04, 2014 09:53:34 AM com.mycompany.MyClass read INFO:
        Opening file data.txt
```

Следует иметь в виду, что время и имена вызывающего класса и метода автоматически включаются в запись. Но если сделать следующий вызов:

```
Logger.global.setLevel(Level.OFF);
```

то вызов метода `info()` не возымеет никакого действия.

НА ЗАМЕТКУ. В приведенном выше примере сообщение **`"Opening file " + filename`** составляется даже в том случае, если протоколирование запрещено. Если же имеют значение затраты на составление подобного сообщения, то ради экономии можно воспользоваться лямбда-выражением следующим образом:

```
Logger.getGlobal().info(() -> "Opening file " + filename);
```

5.3.2. Регистраторы

В профессионально написанной прикладной программе все записи не протоколируются в одном глобальном регистраторе. Вместо этого можно определить свои регистраторы.

Если регистратор запрашивается по заданному имени в первый раз, то он создается. А при последующих вызовах по тому же самому имени предоставляется один и тот же объект регистратора, как показано ниже.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

Аналогично именам пакетов, имена регистраторов носят иерархический характер. В действительности регистраторы в большей степени иерархические, чем пакеты. Между пакетом и его родителем отсутствует семантическая взаимосвязь, но родители и потомки регистраторов разделяют некоторые общие свойства. Так, если отключить запись сообщений в регистраторе `"com.mycompany"`, то прекратят свое действие и порожденные регистраторы.

5.3.3. Уровни протоколирования

Имеются семь уровней протоколирования: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`. По умолчанию протоколирование, по существу, происходит на трех верхних уровнях. Впрочем, ничто не мешает установить другой порог протоколирования, как показано ниже. Теперь протоколирование происходит, начиная с уровня `FINE` и выше.

```
logger.setLevel(Level.FINE);
```

С помощью метода `Level.ALL()` можно разрешить протоколирование на всех уровнях, а с помощью метода `Level.OFF()` — запретить его на всех уровнях. Имеются также методы протоколирования, соответствующие каждому уровню, как, например:

```
logger.warning(message);
logger.fine(message);
```

и т.д. С другой стороны, можно вызвать метод `log()`, чтобы предоставить нужный уровень протоколирования следующим образом:

```
Level level = ...;
logger.log(level, message);
```

СОВЕТ. В конфигурации протоколирования по умолчанию все записи протоколируются на уровне **INFO** и выше. Следовательно, для протоколирования отладочных сообщений, удобных для целей диагностики, но малопригодных для пользователя, можно выбрать уровни **CONFIG**, **FINE**, **FINER** и **FINEST**.

ВНИМАНИЕ. Если установить более высокий уровень подробного протоколирования, чем **INFO**, то придется внести соответствующие коррективы в конфигурацию обработчика протоколов. По умолчанию в обработчике протоколов подавляются сообщения, протоколируемые на уровне ниже **INFO**. Подробнее об этом — в разделе 5.3.6.

5.3.4. Другие методы протоколирования

Для отслеживания последовательности выполнения программы имеются следующие служебные методы:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

Так, в следующем примере кода эти методы вызываются для формирования протокольных записей на уровне FINER, начиная со строк ENTRY и RETURN.

```
public int read(String file, String pattern) {
  logger.entering("com.mycompany.mylib.Reader", "read",
        new Object[] { file, pattern });
  ...
  logger.exiting("com.mycompany.mylib.Reader", "read", count);
  return count;
}
```

НА ЗАМЕТКУ. Как ни странно, эти методы так и не были превращены в методы с переменным количеством аргументов.

Протоколирование чаще всего применяется для регистрации исключений. Описание исключения в протокольной записи принимают два приведенных ниже служебных метода.

```
void log(Level l, String message, Throwable t)
void throwing(String className, String methodName, Throwable t)
```

Ниже приведены типичные примеры их применения. В результате вызова метода throwing() на уровне FINER протоколируется запись и сообщение, начинающееся со строки THROW.

```
try {
  ...
}
  catch (IOException ex) {
  logger.log(Level.SEVERE, "Cannot read configuration", ex);
}
```

и

```
if (...) {
  IOException ex = new IOException("Cannot read configuration");
  logger.throwing("com.mycompany.mylib.Reader", "read", ex);
  throw ex;
}
```

НА ЗАМЕТКУ. В протокольной записи по умолчанию отображается имя класса и метода, содержащего вызов для протоколирования, выводимый из стека вызовов. Но если виртуальная машина оптимизирует исполнение кода, то точные сведения о вызове могут быть недоступны. Чтобы предоставить точное местоположение вызывающего класса или метода, можно воспользоваться методом **logp()**, сигнатура которого приведена ниже.

```
void logp(Level l, String className, String methodName, String message)
```

НА ЗАМЕТКУ. Если протокольные сообщения требуется сделать понятными пользователям на многих языках, их можно локализовать с помощью следующих методов:

```
void logrb(Level level, String sourceClass, String sourceMethod,
  ResourceBundle bundle, String msg, Object... params)
void logrb(Level level, String sourceClass, String sourceMethod,
  ResourceBundle bundle, String msg, Throwable thrown)
```

Комплекты ресурсов рассматриваются в главе 13.

5.3.5. Конфигурация протоколирования

Отредактировав файл конфигурации, можно изменить различные свойства системы протоколирования. Используемый по умолчанию файл конфигурации находится в каталоге `jre/lib/logging.properties`. Чтобы воспользоваться другим файлом, достаточно установить в свойстве `java.util.logging.config.file` местоположение файла конфигурации, начиная с прикладной программы, следующим образом:

```
java -Djava.util.logging.config.file=configFile MainClass
```

ВНИМАНИЕ. Вызов метода **System.setProperty("java.util.logging.config.file", configFile)** в методе **main()** не возымеет никакого действия, поскольку диспетчер протоколирования инициализируется при запуске виртуальной машины и до выполнения метода **main()**.

Чтобы изменить уровень протоколирования по умолчанию, необходимо внести коррективы в следующей строке файла конфигурации:

```
.level=INFO
```

Кроме того, уровни протоколирования для своих регистраторов можно указать, введя в файл конфигурации строки, аналогичные приведенной ниже. По существу, это означает присоединить суффикс **.level** к имени регистратора.

```
com.mycompany.myapp.level=FINE
```

Как будет показано в следующем разделе, регистраторы на самом деле не выводят сообщения на консоль, поскольку это задача обработчиков протоколов, у которых также имеются свои уровни. Так, чтобы вывести на консоль сообщения, регистрируемые на уровне FINE, нужно также установить следующую строку в файле конфигурации.

```
ava.util.logging.ConsoleHandler.level=FINE
```

ВНИМАНИЕ. Настройки в файле конфигурации диспетчера протоколирования не относятся к системным свойствам. В частности, запуск программы из командной строки с параметром **-Dcom.mycompany.myapp.level=FINE** не возымеет никакого действия на регистратор.

Уровни протоколирования можно изменить и при выполнении прикладной программы, используя утилиту **jconsole**. Подробнее об этом можно узнать по адресу www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl.

5.3.6. Обработчики протоколов

По умолчанию регистраторы посылают записи консольному обработчику типа ConsoleHandler, который направляет их в стандартный поток вывода ошибок System.err. В частности, регистратор посылает запись родительскому обработчику, а у окончательного родителя (с именем "") имеется обработчика типа ConsoleHandler.

У обработчиков, как и у регистраторов, имеются свои уровни протоколирования. Так, для протоколирования записи ее уровень должен превышать порог как регистратора, так и обработчика. Уровень протоколирования для консольного обработчика, используемого по умолчанию, устанавливается в файле конфигурации диспетчера протоколирования следующим образом:

```
java.util.logging.ConsoleHandler.level=INFO
```

Чтобы зарегистрировать записи на уровне FINE, следует изменить как уровень регистратора, устанавливаемый по умолчанию, так и уровень обработчика в файле конфигурации. С другой стороны, можно вообще не обращаться к файлу конфигурации, установив свой обработчик, как показано ниже.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

По умолчанию регистратор посылает записи как своим обработчикам, так и обработчикам своего родителя. Так, приведенный выше обработчик является производным от окончательного родителя "", посылающего все записи уровня **INFO** и выше на консоль. Но эти записи не должны выводиться дважды, и поэтому в свойстве `useParentHandlers` устанавливается логическое значение `false`.

Чтобы направить записи в какое-нибудь другое место, следует ввести еще один обработчик. С этой целью в прикладном программном интерфейсе API для протоколирования предоставляются следующие два класса обработчиков: `FileHandler` и `SocketHandler`. В частности, обработчик типа `SocketHandler` посылает записи на указанный хост или порт. Еще больший интерес представляет обработчик типа `FileHandler`, накапливающий записи в файле. В этом случае записи можно просто направить обработчику файлов по умолчанию следующим образом:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

Записи посылаются в файл `javan.log`, находящийся в начальном каталоге пользователя, где *n* — номер, однозначно определяющий файл. По умолчанию записи составляются в формате XML. Типичная протокольная запись имеет следующую форму:

```
<record>
  <date>2014-08-04T09:53:34</date>
  <millis>1407146014072</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.horstmann.corejava.Employee</class>
  <method>read</method>
  <thread>10</thread>
  <message>Opening file staff.txt</message>
</record>
```

Стандартное поведение обработчика файлов можно изменить, установив различные параметры в файле конфигурации диспетчера протоколирования (как поясняется в табл. 5.1) или воспользовавшись одним из следующих конструкторов:

```
FileHandler(String pattern)
FileHandler(String pattern, boolean append)
FileHandler(String pattern, int limit, int count)
FileHandler(String pattern, int limit, int count, boolean append)
```

Таблица 5.1. Назначение параметров конфигурации обработчика файлов

Свойство конфигурации	Описание	Значение по умолчанию
`java.util.logging.FileHandler.level`	Уровень обработчика	`Level.ALL`
`java.util.logging.FileHandler.append`	Если это свойство принимает логическое значение **true**, протокольные записи в состоянии **false** присоединяются в конце файла, а иначе при каждом запуске программы на выполнение открывается новый файл	**false**

Окончание табл. 5.1

Свойство конфигурации	Описание	Значение по умолчанию
`java.util.logging.FileHandler.limit`	Приблизительное максимальное количество байтов, записываемых в файл, прежде чем открыть другой файл (нулевое значение обозначает отсутствие ограничения)	0 в классе `FileHandler`, 50000 в файле конфигурации диспетчера протоколирования по умолчанию
`java.util.logging.FileHandler.pattern`	Шаблон имени файла протокола `%h/java%u.log` (см. далее табл. 5.2)	`%h/java%u.log`
`java.util.logging.FileHandler.count`	Количество протоколов в последовательности циклической смены	1 (циклическая смена отсутствует)
`java.util.logging.FileHandler.filter`	Фильтр для фильтрации протокольных записей (см. далее раздел 5.3.7)	Фильтрация отсутствует
`java.util.logging.FileHandler.encoding`	Кодировка символов	Кодировка символов на конкретной платформе
`java.util.logging.FileHandler.formatter`	Средство форматирования каждой протокольной записи	`java.util.logging.XMLFormatter`

Имя файла протокола, выбираемое по умолчанию, обычно не употребляется. Вместо этого применяется шаблон `%h/myapp.log`, переменные в котором поясняются в табл. 5.2.

Таблица 5.2. Переменные из шаблона имени файла протокола

Переменная	Описание
`%h`	Начальный каталог пользователя (свойство `user.hom`)
`%t`	Временный каталог для системы
`%u`	Однозначный номер файла
`%g`	Номер версии для циклически сменяемых протоколов (суффикс `.%g` употребляется в том случае, если указана циклическая смена протоколов, а переменная `%g` в шаблоне отсутствует)
`%%`	Знак процентов

Если один и тот же файл протокола используется несколькими прикладными программами (или несколькими копиями одной и той же прикладной программы), то в свойстве `append` следует установить логическое значение `true`. С другой стороны, в шаблоне имени файла протокола можно указать переменную `%u`, чтобы каждая прикладная программа создавала однозначную копию протокола.

Рекомендуется также включить режим циклической смены файлов протоколов. В этом режиме файлы протоколов хранятся в последовательности циклической смены вроде следующей: `myapp.log.0`, `myapp.log.1`, `myapp.log.2` и т.д. Всякий раз, когда размер файла протокола превышает заданный предел, старый файл удаляется, а остальные файлы переименовываются, и создается новый файл с номером версии 0.

5.3.7. Фильтры и средства форматирования

Помимо фильтрации по уровням протоколирования, каждый регистратор и обработчик может иметь дополнительный фильтр, реализующий функциональный интерфейс `Filter` со следующим единственным методом:

```
boolean isLoggable(LogRecord record)
```

Чтобы установить фильтр в регистраторе или обработчике, следует вызвать метод `setFilter()`. Следует, однако, иметь в виду, что одновременно можно пользоваться только одним фильтром.

Классы `ConsoleHandler` и `FileHandler` формируют протокольные записи в текстовом виде и в формате XML. Но можно определить и свои форматы, расширив класс `Formatter` и переопределив метод `String format(LogRecord record)`. Запись можно отформатировать как угодно и возвратить результирующую символьную строку. В методе форматирования, возможно, придется вызвать метод `String formatMessage(LogRecord record)`. Этот метод форматирует ту часть записи, которая относится к сообщению, подставляя параметры и выполняя локализацию.

Многие форматы файлов (в том числе XML) требуют наличия заголовочной и хвостовой частей вокруг отформатированных записей. С этой целью придется переопределить приведенные ниже методы. И наконец, для установки средства форматирования в обработчике следует вызвать метод `setFormatter()`.

```
String getHead(Handler h)
String getTail(Handler h)
```

Упражнения

1. Напишите метод `public ArrayList<Double> readValues(String filename) throws ...`, читающий числа с плавающей точкой из файла. Сгенерируйте подходящие исключения, если файл не удается открыть или же если введены данные, не относящиеся к числам с плавающей точкой.
2. Напишите метод `public double sumOfValues(String filename) throws ...`, вызывающий метод из предыдущего упражнения и возвращающий сумму значений в файле. Организуйте распространение любых исключений вызывающему коду.
3. Напишите программу, вызывающую метод из предыдущего упражнения и выводящую полученный результат. Организуйте перехват исключений и предоставьте ответную реакцию на действия пользователя в виде сообщений о любых ошибочных условиях.
4. Повторите предыдущее упражнение, но на этот раз не пользуйтесь исключениями. Вместо этого организуйте возврат кодов ошибок из методов `readValues()` и `sumOfValues()`.

5. Реализуйте метод, содержащий код, где применяются классы `Scanner` и `PrintWriter` (см. раздел 5.1.5). Но вместо оператора `try` с ресурсами воспользуйтесь оператором `catch`. Непременно закройте оба объекта, при условии, что они построены надлежащим образом. Для этого вам придется принять во внимание следующие условия.
   - Конструктор класса `Scanner` генерирует исключение.
   - Конструктор класса `PrintWriter` генерирует исключение.
   - Метод `hasNext()`, `next()` или `println()` генерирует исключение.
   - Метод `out.close()` генерирует исключение.
   - Метод `in.close()` генерирует исключение.
6. В разделе 5.1.6 приведен пример ошибочного внедрения блока операторов `catch` и `finally` в блок кода оператора `try`. Исправьте этот код, во-первых, перехватив исключение в операторе `finally`, во-вторых, внедрив блок операторов `try/finally` в блок операторов `try/catch`, и в-третьих, применив оператор `try` с ресурсами вместе с оператором `catch`.
7. Для выполнения этого упражнения вам придется просмотреть исходный код класса `java.util.Scanner`. Если при использовании класса `Scanner` операция ввода завершается неудачно, в этом классе перехватывается исключение в связи с ошибками ввода данных и закрывается ресурс, из которого осуществляется ввод. Что произойдет, если при закрытии ресурса будет сгенерировано исключение? Каким образом эта реализация взаимодействует с обработкой подавляемых исключений в операторе `try` с ресурсами?
8. Разработайте вспомогательный метод, чтобы воспользоваться классом `ReentrantLock` в операторе `try` с ресурсами. Вызовите метод `lock()` и возвратите объект класса, который реализует интерфейс `AutoCloseable` и в методе которого `close()` вызывается метод `unlock()`, но не генерируется никаких исключений.
9. Методы из классов `Scanner` и `PrintWriter` не генерируют проверяемые исключения, чтобы ими легче было пользоваться начинающим программировать на Java. Как же тогда выяснить, произошли ли ошибки в процессе чтения или записи данных? Следует иметь в виду, что конструкторы *могут* генерировать исключения. Почему это способно лишить всякого смысла цель упрощения классов для начинающих программировать на Java?
10. Напишите рекурсивный метод `factorial()`, выводящий все кадры стека перед возвратом значения. Постройте (но не генерируйте) объект исключения любого типа и получите результат трассировки его стека, как пояснялось в разделе 5.1.8.
11. Сравните вызов `Objects.requireNonNull(obj)` с утверждением `assert obj != null`. Приведите убедительные примеры применения того и другого.
12. Напишите метод `int min(int[] values)`, в котором перед возвратом наименьшего значения утверждается, что оно действительно не больше всех остальных

значений в массиве. Воспользуйтесь вспомогательным методом или же методом `Stream.allMatch()`, если вы уже просматривали материал главы 8. Организуйте повторный вызов данного метода для обработки крупного массива и определите время выполнения кода при разрешении, запрете и исключении утверждений.

13. Реализуйте и испытайте фильтр протокольных записей, содержащий такие неприличные слова, как секс, наркотики и C++.

14. Реализуйте и испытайте средство форматирования протокольных записей, создающее HTML-файл.

Глава 6

Обобщенное программирование

В этой главе...

Нередко требуется реализовать классы и методы, способные обрабатывать разнотипные данные. Например, в списочном массиве `ArrayList<T>` могут храниться элементы произвольного типа `T`. В таком случае класс `ArrayList` называется *обобщенным*, а `T` — *параметром типа*. Основной принцип обобщений довольно прост и делает их очень удобными. В первых двух разделах данной главы будет рассмотрена более простая часть этого принципа.

В любом языке программирования с обобщенными типами подробности реализации усложняются по мере ограничения или варьирования параметров типа. Допустим, что требуется отсортировать элементы массива. В таком случае нужно указать, что обобщенный тип `T` обеспечивает упорядочение. А если параметр типа варьируется, то что это должно означать для обобщенного типа? Какой, например, должна быть взаимосвязь между классом `ArrayList<String>` и методом, ожидающим объект типа `ArrayList<Object>`? В разделах 6.3 и 6.4 поясняется, каким образом в Java решаются эти вопросы.

Обобщенное программирование на Java оказывается сложнее, чем должно быть, поскольку обобщения были внедрены в Java не сразу, а со временем, причем это было сделано с учетом обратной совместимости. Как следствие, появился целый ряд прискорбных ограничений. Одни их из них непосредственно затрагивают интересы всех, кто программирует на Java, а другие — только интересы разработчиков обобщенных классов. Подробнее об этом речь пойдет в разделах 6.5 и 6.6. И в последнем разделе этой главы обобщения рассматриваются в связи с рефлексией. Если вы не пользуетесь рефлексией в своих программах, то можете пропустить этот раздел.

Основные положения этой главы приведены ниже.

1. Обобщенным называется класс с одним и больше параметром типа.
2. Обобщенным называется метод с параметрами типа.
3. Параметр типа может обозначать подтип одного или нескольких типов.
4. Обобщенные типы инвариантны. Если **`S`** — подтип **`T`**, то между типами `G<S>` и `G<T>` не существует никакой взаимосвязи.
5. С помощью метасимволов подстановки `G<? extends T>` или `G<? super T>` можно указать, что метод способен принимать реализацию обобщенного типа с аргументом подкласса или суперкласса.
6. Параметры типа стираются при компиляции обобщенных классов и методов.
7. Стирание накладывает немало ограничений на обобщенные типы. В частности, нельзя получать экземпляры обобщенных классов или массивов, выполнять приведение к обобщенному типу или генерировать объект обобщенного типа.
8. Класс `Class<T>` является обобщенным, что удобно, поскольку методы вроде `newInstance()` объявляются для получения значения типа `T`.
9. Несмотря на то что обобщенные классы и методы стираются в виртуальной машине, во время выполнения можно выяснить, каким образом они были объявлены.

6.1. Обобщенные классы

Обобщенным называется класс с одним или больше *параметром типа*. В качестве простого примера рассмотрим следующий обобщенный класс для хранения пар "ключ–значение":

```
public class Entry<K, V> {
  private K key;
  private V value;

  public Entry(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey() { return key; }
  public V getValue() { return value; }
}
```

Как видите, параметры типа `K` и `V` указаны в угловых скобках после имени класса. А в определениях членов классов они используются для обозначения типов переменных экземпляра, параметров методов и возвращаемых значений.

Для получения экземпляра обобщенного класса вместо обобщенных типов переменных подставляются конкретные типы. Например, `Entry<String, Integer>` — это обыкновенный класс с методами `String getKey()` и `Integer getValue()`.

ВНИМАНИЕ. Параметры типа нельзя заменить примитивными типами. Например, выражение `Entry<String, int>` в Java недействительно.

При *построении* объекта обобщенного класса нельзя опускать параметры типа в конструкторе, как показано в следующем примере кода:

```
Entry<String, Integer> entry = new Entry<>("Fred", 42);
  // То же, что и new Entry<String, Integer>("Fred", 42)
```

Обратите внимание на то, что пустая пара угловых скобок по-прежнему требуется перед аргументами конструктора. Некоторые называют такую пару *ромбовидной*. Если используется ромбовидный синтаксис, то параметры типа автоматически выводятся для конструктора.

6.2. Обобщенные методы

Как и обобщенный класс, *обобщенный метод* имеет *параметры типа*. Обобщенным может быть метод из обычного или обобщенного класса. Ниже приведен характерный пример обобщенного метода, объявляемого в классе, который не является обобщенным.

```
public class Arrays {
  public static <T> void swap(T[] array, int i, int j) {
```

```
    T temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }
}
```

Метод `swap()` служит для перестановки элементов в произвольном массиве, при условии, что они не относятся к примитивному типу:

```
String[] friends = ...;
Arrays.swap(friends, 0, 1);
```

При объявлении обобщенного метода параметр типа указывается после модификаторов доступа (как, например, `public` и `static` в приведенном ниже примере) и перед возвращаемым типом.

```
public static <T> void swap(T[] array, int i, int j)
```

Вызывая обобщенный метод, совсем не обязательно указывать параметр типа. Он автоматически выводится из типов параметров метода и возвращаемого типа. Например, в вызове `Arrays.swap(friends, 0, 1)` параметр `friends` относится к типу `String[]`, и поэтому компилятор может автоматически сделать вывод, что обобщенный тип **`T`** должен быть конкретным типом `String`. При желании этот тип можно указать явно перед именем метода, как в приведенном ниже примере вызова. Но этого не стоит делать, в частности, для того, чтобы получать более подробные сообщения об ошибках, когда что-нибудь не заладится в программе (см. упражнение 5 в конце этой главы).

```
Arrays.<String>swap(friends, 0, 1);
```

Прежде чем вдаваться в технические подробности обобщений в последующих разделах, стоит еще раз бросить взгляд на приведенные выше примеры обобщенного класса `Entry` и метода `swap()`, чтобы оценить, насколько удобны и естественны обобщенные типы. Так, в классе `Entry` произвольными могут быть типы ключей и значений, а в методе `swap()` — тип массива. Это ясно и просто выражается параметрами типа.

6.3. Ограничения типов

Иногда параметры типа обобщенного класса или метода должны удовлетворять определенным требованиям. В качестве такого требования можно, в частности, указать *ограничение типа*, чтобы этот тип расширял определенные классы или реализовывал определенные интерфейсы.

Допустим, что имеется списочный массив типа `ArrayList` объектов класса, реализующего интерфейс `AutoCloseable`, как показано ниже. Эти объекты требуется вовремя закрыть.

```
public static <T extends AutoCloseable> void closeAll(ArrayList<T> elems)
      throws Exception {
```

```
  for (T elem : elems) elem.close();
}
```

Ограничение типа `extends AutoCloseable` гарантирует, что тип элементов списочного массива является подтипом, производным от интерфейса `AutoCloseable`. Следовательно, вызов `elem.close()` является достоверным. Этому методу можно передать списочный массив типа `ArrayList<PrintStream>`, но не типа `ArrayList<String>`. Следует иметь в виду, что ключевое слово `extends` в ограничении типа фактически означает "подтип". Разработчики Java просто воспользовались ключевым словом `extends`, вместо того чтобы изобретать еще одно ключевое слово или знак. Более интересный вариант данного метода упоминается в упражнении 14 в конце этой главы.

НА ЗАМЕТКУ. В рассматриваемом здесь примере ограничение типа требуется потому, что параметр метода относится к типу **`ArrayList`**. Если бы данный метод принимал массив, то такой метод совсем не обязательно было бы объявлять обобщенным. Вместо этого данный метод можно было бы объявить как обычный следующим образом:

```
public static void closeAll(AutoCloseable[] elems) throws Exception
```

Это вполне допустимо, поскольку тип массива вроде **`PrintStream[]`** является подтипом, производным от типа **`AutoCloseable[]`**. Но, как будет показано в следующем разделе, тип **`ArrayList<PrintStream>`** *не* является подтипом, производным от типа **`ArrayList <AutoCloseable>`**. Это затруднение разрешает параметр ограниченного типа.

Параметр типа может иметь несколько ограничений, как показано ниже. Этот синтаксис такой же, как и при перехвате нескольких исключений.

```
T extends Runnable & AutoCloseable
```

Ограничений на интерфейсы можно наложить сколько угодно, но хотя бы одно из них должно быть наложено на класс. Если требуется наложить ограничение на класс, то оно должно быть указано первым в списке ограничений.

6.4. Вариантность типов и метасимволы подстановки

Допустим, что требуется реализовать метод, обрабатывающий массив объектов, относящихся к подклассам, производным от класса `Employee`. Для этого достаточно объявить его с параметром типа `Employee[]` следующим образом:

```
public static void process(Employee[] staff) { ... }
```

Так, если класс `Manager` является подклассом, производным от класса `Employee`, то данному методу можно передать массив типа `Manager[]`, поскольку последний относится к подтипу, производному от типа `Employee[]`. Такое поведение называется *ковариантностью*. Массивы изменяются в той же мере, в какой и типы их элементов.

А теперь допустим, что требуется обработать элементы списочного массива. Но дело в том, что тип `ArrayList<Manager>` *не* является подтипом, производным от типа `ArrayList<Employee>`. И для такого ограничения имеются веские основания. Если бы списочный массив типа `ArrayList<Manager>` допускалось присваивать перемен-

ной типа ArrayList<Employee>, то его можно было бы легко испортить, попытавшись сохранить в нем сведения о сотрудниках, не относящихся к руководящему составу, как показано ниже. Преобразование типа ArrayList<Manager> в тип ArrayList<Employee> не разрешается, и поэтому подобная ошибка произойти не может.

```
ArrayList<Manager> bosses = new ArrayList<>();
ArrayList<Employee> empls = bosses; // Нельзя, но допустим, что можно ...
empls.add(new Employee(...)); // Простой сотрудник среди начальства!
```

НА ЗАМЕТКУ. Может ли подобная ошибка возникнуть в тех массивах, где разрешается преобразование типа **ArrayList<Manager>** в тип **ArrayList<Employee>**? Безусловно, может, как было показано в главе 4. Ведь в Java массивы ковариантны, что, конечно, удобно, но ненадежно. Так, если объект типа **Employee**, представляющий простого сотрудника, сохраняется в массиве типа **Manager[]**, где хранятся сведения о руководящем составе, то генерируется исключение типа **ArrayStoreException**. С другой стороны, все обобщенные типы в Java являются *инвариантными*.

Для указания порядка варьирования параметров метода и возвращаемого типа в Java употребляются *метасимволы подстановки*. Этот механизм иногда еще называют *используемой по месту вариантностью*. Подробнее об этом речь пойдет в последующих разделах.

6.4.1. Метасимволы подстановки подтипов

Во многих случаях взаимное преобразование списочных массивов оказывается совершенно безопасным. Так, если метод вообще не записывает данные в массив, то его аргумент не может быть испорчен. Это обстоятельство можно выразить с помощью метасимвола подстановки следующим образом:

```
public static void printNames(ArrayList<? extends Employee> staff) {
  for (int i = 0; i < staff.size(); i++) {
    Employee e = staff.get(i);
    System.out.println(e.getName());
  }
}
```

Подстановочный тип ? extends Employee обозначает некоторый неизвестный подтип, производный от типа Employee. Такой метод можно вызвать с параметром типа ArrayList<Employee> или другим списочным массивом соответствующего подтипа, например ArrayList<Manager>.

Метод get() из класса ArrayList<? extends Employee> должен возвращать тип ? extends Employee, поэтому следующее выражение вполне допустимо:

```
Employee e = staff.get(i);
```

Что бы ни обозначал тип ?, он является подтипом, производным от типа Employee, а результат вызова staff.get(i) может быть присвоен переменной e от типа Employee. (Цикл for был специально использован в рассматриваемом здесь примере, чтобы показать, каким образом элементы вызываются из списочного массива.)

Что же произойдет, если попытаться сохранить элемент в списочном массиве типа ArrayList<? extends Employee>? Подобная попытка окажется неудачной. Рассмотрим следующий вызов:

```
staff.add(x);
```

У метода add() имеется параметр типа ? extends Employee, но *не существует* такого объекта, который можно было бы передать этому методу. Так, если попытаться передать ему объект типа Manager, компилятор откажет в этом. Ведь подстановочный тип ? может обозначать *любой* подкласс (например, Janitor), а следовательно, объект типа Manager нельзя ввести в списочный массив типа ArrayList<Janitor>.

НА ЗАМЕТКУ. Данному методу можно, конечно, передать пустое значение **null**, но ведь это не объект.

В целом подстановочный тип ? extends Employee можно преобразовать в тип Employee, но ни один из типов — в подстановочный тип ? extends Employee. Именно поэтому читать данные из списочного массива типа ArrayList<? extends Employee> можно, но записывать в него данные нельзя.

6.4.2. Метасимволы подстановки супертипов

Подстановочный тип ? extends Employee обозначает произвольный подтип, производный от типа Employee. С другой стороны, подстановочный тип ? super Employee обозначает супертип для типа Employee. Такие подстановочные типы зачастую оказываются удобными в качестве параметров в функциональных объектах. В качестве примера рассмотрим следующий интерфейс Predicate с единственным методом, где проверяется, имеется ли у объекта типа **T** конкретное свойство:

```
public interface Predicate<T> {
  boolean test(T arg);
  ...
}
```

Следующий метод выводит имена всех работников с заданным свойством:

```
public static void printAll(Employee[] staff, Predicate<Employee> filter) {
  for (Employee e : staff)
    if (filter.test(e))
      System.out.println(e.getName());
}
```

Этот метод можно вызвать с объектом типа Predicate<Employee>. А поскольку этот тип обозначает функциональный интерфейс, то можно передать и лямбда-выражение, как показано ниже.

```
printAll(employees, e -> e.getSalary() > 100000);
```

А теперь допустим, что требуется воспользоваться функциональным интерфейсом Predicate<Object>, например, следующим образом:

```
printAll(employees, e -> e.toString().length() % 2 == 0);
```

На первый взгляд, это не должно вызвать никаких трудностей. Ведь каждый объект типа Employee наследует метод toString() от класса Object. Но, как и все остальные обобщенные типы, функциональный интерфейс Predicate инвариантен, а следовательно, взаимосвязь между типами Predicate<Employee> и Predicate<Object> отсутствует. В качестве выхода из этого положения можно указать подстановочный тип Predicate<? super Employee> следующим образом:

```
public static void printAll(Employee[] staff,
       Predicate<? super Employee> filter) {
  for (Employee e : staff)
    if (filter.test(e))
      System.out.println(e.getName());
}
```

Рассмотрим подробнее вызов метода filter.test(e). Параметр метода test() относится к типу, служащему некоторым супертипом для типа Employee. Подобная ситуация типична, когда функции естественным образом *контравариантны* в отношении типов их параметров. Например, когда предполагается, что функция может обрабатывать данные о работниках, вполне допустимо предоставить такой ее вариант, который способен обрабатывать произвольные объекты. В общем, если в качестве параметра метода указывается обобщенный функциональный интерфейс, следует использовать подстановочный тип super.

НА ЗАМЕТКУ. Некоторые программисты предпочитают для указания метасимволов подстановки мнемонику PECS: ключевое слово **extends** для поставщика и ключевое слово **super** для потребителя. Например, списочный массив типа **ArrayList**, из которого читаются данные, является потребителем, и поэтому подстановочный тип указывается с ключевым словом **extends**. А интерфейс **Predicate**, которому передаются данные для проверки, является потребителем, и поэтому подстановочный тип указывается с ключевым словом **super**.

6.4.3. Применение метасимволов подстановки в переменных типа

Рассмотрим обобщение метода из предыдущего примера. В данном случае из массива выводятся элементы, удовлетворяющие некоторому условию:

```
public static <T> void printAll(T[] elements, Predicate<T> filter) {
  for (T e : elements)
   if (filter.test(e))
      System.out.println(e.toString());
}
```

Этот обобщенный метод пригоден для массивов любого типа. Его параметр типа обозначает тип передаваемого ему массива. Но такой метод страдает ограничением, упоминавшимся в предыдущем разделе, а именно: параметр типа Predicate должен точно соответствовать параметру типа метода.

Преодолеть данное ограничение можно таким же образом, как и прежде, но на этот раз ограничение метасимволом подстановки накладывается на переменную

типа. Так, следующий метод принимает в качестве параметра фильтр элементов типа **T** или любой его супертип:

```
public static <T> void printAll(T[] elements, Predicate<? super T> filter)
```

Рассмотрим еще один пример. Интерфейс `Collection<E>`, подробнее рассматриваемый в следующей главе, описывает коллекцию элементов типа `E`. У него имеется следующий метод:

```
public void addAll(Collection<? extends E> c)
```

Этот метод позволяет ввести все элементы из другой коллекции, элементы которой также относятся к типу **E** или некоторому его подтипу. В частности, этот метод дает возможность ввести коллекцию руководителей в коллекцию работников, но не наоборот.

Чтобы показать, насколько сложными могут быть объявления типов, рассмотрим следующее определение метода `Collections.sort()`:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Интерфейс `List`, подробнее рассматриваемый в следующей главе, описывает последовательность элементов вроде связного списка или списочного массива типа `ArrayList`. А метод `sort()` способен отсортировать любой список типа `List<T>`, при условии, что **T** — это подтип, производный от типа `Comparable`. Но ведь интерфейс `Comparable` также является обобщенным, как показано ниже.

```
public interface Comparable<T> {
  int compareTo(T other);
}
```

Его параметр типа обозначает тип аргумента метода `compareTo()`. Следовательно, метод `Collections.sort()` можно было бы объявить следующим образом:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

Но такое объявление было бы слишком ограниченным. Допустим, что класс `Employee` реализует интерфейс `Comparable<Employee>`, предназначенный для сравнения работников по зарплате. Допустим также, что класс `Manager` расширяет класс `Employee`. Следует, однако, иметь в виду, что он реализует интерфейс `Comparable<Employee>`, а *не* интерфейс `Comparable<Manager>`. Следовательно, класс `Manager` является подтипом, производным *не* от типа `Comparable<Manager>`, а от типа `Comparable<? super Manager>`.

НА ЗАМЕТКУ. В некоторых языках программирования (например, C# и Scala) параметры типа можно объявлять ковариантными или контравариантными. Например, объявив параметр типа **Comparable** как контравариантный, совсем не обязательно употреблять метасимвол подстановки для указания каждого параметра типа **Comparable**. Такая вариантность по месту объявления удобна, но менее эффективна, чем используемая по месту вариантность подстановочных типов, обозначаемых метасимволами в Java.

6.4.4. Неограниченные метасимволы подстановки

Неограниченные метасимволы подстановки можно применять в тех случаях, когда выполняются только весьма обобщенные операции. В качестве примера ниже приведен метод, в котором проверяется, содержит ли списочный массив типа `ArrayList` какие-нибудь пустые элементы.

```
public static boolean hasNulls(ArrayList<?> elements) {
  for (Object e : elements) {
    if (e == null) return true;
  }
  return false;
}
```

Параметр типа для класса `ArrayList` особого значения не имеет, и поэтому в данном случае целесообразно употребить подстановочный тип `ArrayList<?>`. С тем же успехом метод `hasNulls()` можно было бы сделать обобщенным, как показано ниже. Но метасимвол подстановки проще понять, и поэтому именно такой подход более предпочтителен.

```
public static <T> boolean hasNulls(ArrayList<T> elements)
```

6.4.5. Захват подстановки

Попробуем определить метод `swap()`, используя символы подстановки следующим образом:

```
public static void swap(ArrayList<?> elements, int i, int j) {
  ? temp = elements.get(i); // Не пройдет!
  elements.set(i, elements.get(j));
  elements.set(j, temp);
}
```

Этот вариант не пройдет. Ведь метасимвол подстановки **?** можно применять как аргумент типа, но не как тип. Но это ограничение можно обойти, введя вспомогательный метод:

```
public static void swap(ArrayList<?> elements, int i, int j) {
  swapHelper(elements, i, j);
}

private static <T> void swapHelper(ArrayList<T> elements, int i, int j) {
  T temp = elements.get(i);
  elements.set(i, elements.get(j));
  elements.set(j, temp);
}
```

Вызов метода `swapHelper()` оказывается достоверным благодаря специальному правилу, называемому *захватом подстановки*. Компилятору неизвестно, что именно обозначает метасимвол подстановки **?**, но ведь он обозначает какой-то тип, и поэтому вполне допустимо вызвать обобщенный метод. Параметр типа **T** в методе `swapHelper()` "захватывает" подстановочный тип. А поскольку метод `swapHelper()`

является обобщенным, а не обычным методом, объявляемым с метасимволами подстановки типов его параметров, то переменную типа `T` можно использовать для объявления других переменных.

Чего же мы этим добились? Пользователь прикладного программного интерфейса API увидит простой для понимания класс `ArrayList<?>` вместо обобщенного метода.

6.5. Обобщения в виртуальной машине

Когда обобщенные классы и методы внедрялись в Java, разработчики этого языка стремились сделать обобщенные формы классов совместимыми с их предыдущими вариантами. Например, класс `ArrayList<String>` можно было бы передать методу из версии, предшествовавшей появлению обобщений, где допускалось принимать класс `ArrayList`, накапливающий элементы типа `Object`. Разработчики Java остановили свой выбор на реализации обобщений, "стирающей" типы в виртуальной машине. В то время это был весьма распространенный механизм, позволявший программирующим на Java постепенно перейти к применению обобщений. Нетрудно догадаться, что у данного механизма имеются свои недостатки, и, как часто бывает с компромиссами, на которые приходится идти ради совместимости, эти недостатки так и не были устранены после успешного завершения перехода к обобщениям. В этом разделе поясняется, что же происходит с обобщениями в виртуальной машине, а в следующем разделе поясняются соответствующие последствия.

6.5.1. Стирание типов

Когда определяется обобщенный тип, он компилируется в *базовый* (так называемый "сырой" тип). Например, класс `Entry<K, V>`, представленный в разделе 6.1, превращается в следующий класс, где обобщенные типы `K` и `V` заменяются типом `Object`:

```
public class Entry {
  private Object key;
  private Object value;

  public Entry(Object key, Object value) {
    this.key = key;
    this.value = value;
  }

  public Object getKey() { return key; }
  public Object getValue() { return value; }
}
```

Если же на параметр типа накладываются ограничения, то его тип заменяется первым ограничением. Допустим, что класс `Entry` объявляется следующим образом:

```
public class Entry<K extends Comparable<? super K> & Serializable,
                   V extends Serializable>
```

В результате стирания типов он превращается в такой класс:

```
public class Entry {
  private Comparable key;
  private Serializable value;
  ...
}
```

6.5.2. Вставка приведения типов

Стирание типов кажется в какой-то мере опасным механизмом, но на самом деле оно совершенно безопасно. Допустим, что используется объект типа `Entry<String, Integer>`. При построении этого объекта нужно предоставить ключ типа `String` и значение типа `Integer` или же преобразовать его в данный тип. В противном случае прикладной код вообще не будет скомпилирован. Таким образом, гарантируется, что метод `getKey()` возвращает объект типа `String`.

Но допустим, что прикладной код компилируется с "необрабатываемыми" предупреждениями, возможно, потому, что в классе `Entry` выполняется приведение типов или употребляются как обобщенные, так и базовые типы. В таком случае в классе `Entry<String, Integer>` можно указать ключ другого типа.

Следовательно, проверки на безопасность необходимы и во время выполнения. Компилятор вставляет приведение типов всякий раз, когда анализирует выражение со стираемым типом. В качестве примера рассмотрим следующий фрагмент кода:

```
Entry<String, Integer> entry = ...;
String key = entry.getKey();
```

Стираемый метод `getKey()` возвращает объект типа `Object`, и поэтому компилятор генерирует код, равнозначный следующей строке кода:

```
String key = (String) entry.getKey();
```

6.5.3. Мостовые методы

В предыдущих разделах были рассмотрены основы того, что делает стирание типов. Этот механизм безопасен и прост, хотя и не совсем. При стирании типов параметров метода и возвращаемого типа компилятору иногда требуется синтезировать так называемые *мостовые методы*. Подробности их реализации знать не обязательно, разве что для того, чтобы узнать, почему мостовой метод появляется в результатах трассировки стека, или же найти более основательное разъяснение одного из самых неясных ограничений, накладываемых на обобщения в Java (см. далее раздел 6.6.6).

Рассмотрим следующий пример кода:

```
public class WordList extends ArrayList<String> {
  public void add(String e) {
    return isBadWord(e) ? false : super.add(e);
  }
  ...
}
```

А теперь рассмотрим приведенный ниже фрагмент кода, в последней строке которого вызывается (стираемый) метод add(Object) из класса ArrayList.

```
WordList words = ...;
ArrayList<String> strings = words; // Преобразование в суперкласс
                                   // вполне допустимо!
strings.add("C++");
```

Вполне обоснованно предположить, что в данном случае динамический поиск метода будет выполнен таким образом, чтобы вызвать метод add() из класса WordList, а не метод add() из класса ArrayList, когда для объекта типа WordList требуется вызвать метод add(). Именно с этой целью компилятор и синтезирует следующий мостовой метод в классе WordList:

```
public void add(Object e) {
  add((String) e);
}
```

В вызове strings.add("C++") происходит обращение к методу add(Object), и поэтому вызывается метод add(String) из класса WordList. Мостовые методы можно также вызывать при варьировании возвращаемого типа. В качестве примера рассмотрим следующий метод:

```
public class WordList extends ArrayList<String> {
  public String get(int i) {
    return super.get(i).toLowerCase();
  }
  ...
}
```

В классе WordList имеются следующие два метода get():

```
String get(int) // Этот метод определен в классе WordList
Object get(int) // Этот метод переопределяет одноименный метод,
                // определенный в классе ArrayList
```

Второй из перечисленных выше методов синтезируется компилятором и вызывает первый метод. Опять же это делается для поддержки механизма динамического поиска методов.

Эти методы имеют одинаковые типы параметров, но разные возвращаемые типы. Реализовать такую пару методов в языке Java нельзя. Но в виртуальной машине метод указывается по имени, типам параметров *и* возвращаемому типу, что дает компилятору возможность сформировать подобную пару методов.

НА ЗАМЕТКУ. Мостовые методы применяются для реализации не только обобщенных типов, но и ковариантных возвращаемых типов. Так, в главе 4 и следующем примере показано, как объявлять метод **clone()** с подходящим возвращаемым типом:

```
public class Employee implements Cloneable {
  public Employee clone() throws CloneNotSupportedException { ... }
}
```

В данном примере у класса **Employee** имеются следующие два метода **clone()**:

```
Employee clone() // Этот метод определен выше
Object clone()   // А этот мостовой метод синтезируется компилятором
```

И в этом случае мостовой метод синтезируется для поддержки механизма динамического поиска методов. Он вызывает первый метод.

6.6. Ограничения, накладываемые на обобщения

Имеется ряд ограничений, накладываемых на применение обобщенных классов и методов в Java. Одни из них просто необычны, а другие — на самом деле неудобны, но большинство из них являются следствием стирания типов. В последующих разделах рассматриваются те ограничения, с которыми чаще всего приходится иметь дело в практике программирования на Java.

6.6.1. Запрет на аргументы примитивных типов

Параметр типа вообще не может иметь примитивный тип. В частности, сформировать списочный массив типа `ArrayList<int>` нельзя. Как было показано ранее, в виртуальной машине предусмотрен только один базовый тип `ArrayList` для хранения элементов типа `Object` в списочном массиве. А тип `int` является примитивным и *не* относится к объектам.

Когда обобщения только появились в Java, они не считались значительным усовершенствованием этого языка. Ведь в конечном счете можно сформировать списочный массив типа `ArrayList<Integer>`, опираясь на автоупаковку. А теперь, когда обобщения нашли более широкое распространение, неудобство данного ограничения стало еще более очевидным. Впрочем, появилось немало функциональных интерфейсов вроде `IntFunction`, `LongFunction`, `DoubleFunction`, `ToIntFunction`, `ToLongFunction`, `ToDoubleFunction`, которые отвечают за унарные функции и три из восьми примитивных типов.

6.6.2. Во время компиляции все типы оказываются базовыми

В виртуальной машине употребляются только базовые типы. Например, во время выполнения нельзя выяснить, содержит ли списочный массив типа `ArrayList` объекты типа `String`. А следующее условие:

```
if (a instanceof ArrayList<String>)
```

считается во время компиляции ошибочным, поскольку такая проверка вообще не может быть выполнена.

Столь же неэффективным, хотя и допустимым оказывается приведение получаемого экземпляра к обобщенному типу, как показано ниже.

```
Object result = ...;
ArrayList<String> list = (ArrayList<String>) result;
  // ПРЕПРЕЖДЕНИЕ: проверяется лишь соответствие полученного
  // результата базовому типу ArrayList
```

Такое приведение типов разрешается потому, что иногда оно просто неизбежно. Если переменная `result` содержит результат довольно общего процесса (например, вызова метода посредством рефлексии; см. главу 4), а его конкретный тип неизвестен компилятору, то программист должен предусмотреть приведение типов. Хотя приведения к типу `ArrayList` или `ArrayList<?>` будет явно недостаточно.

Чтобы подавить предупреждение, достаточно снабдить объявление переменной `result` аннотацией следующим образом:

```
@SuppressWarnings("unchecked") ArrayList<String> list =
    (ArrayList<String>) result;
```

ВНИМАНИЕ. Злоупотребление аннотацией **`@SuppressWarnings`** может привести к так называемому *загрязнению "кучи"*, когда объекты, которые должны принадлежать конкретной реализации обобщенного типа, на самом деле принадлежат другой реализации. Так, если присвоить списочный массив типа **`ArrayList<Employee>`** ссылке на списочный массив типа **`ArrayList<String>`**, то при извлечении элемента неверного типа возникнет исключение типа **`ClassCastException`**.

СОВЕТ. Главная неприятность, которую доставляет загрязнение "кучи", состоит в том, что ошибка, сообщаемая во время выполнения, совершенно не отражает ее причину: ввод в списочный массив элемента неверного типа. Для устранения подобного недостатка можно воспользоваться так называемым "проверяемым представлением". Для этого строку кода, в которой был построен объект, например, типа **`ArrayList<String>`**, нужно заменить следующей строкой кода:

```
List<String> strings =
    Collections.checkedList(new ArrayList<>(), String.class);
```

Проверяемое представление позволяет контролировать все операции ввода элементов в списочный массив. И если в него вводится объект неверного типа, то генерируется соответствующее исключение.

Метод `getClass()` всегда возвращает базовый тип. Так, если переменная `list` относится к типу `ArrayList<String>`, то в результате вызова `list.getClass()` возвращается *литерал класса* `ArrayList.class`. На самом деле такого *литерала класса*, как `ArrayList<String>.class`, не существует, и поэтому данное выражение считается синтаксической ошибкой. Кроме того, в *литералах классов* нельзя указывать обобщенные типы. Таких *литералов классов*, как, например, `T.class`, `T[].class` или `ArrayList<T>.class`, не существует.

6.6.3. Нельзя получить экземпляры обобщенных типов

Обобщенные типы нельзя употреблять в таких выражениях, как, например, `new T(...)` или `new T[...]`. Подобные формы запрещены, поскольку они не воплощают того, что намеревается сделать программист, когда стирается обобщенный тип **`T`**.

Если нужно получить обобщенный экземпляр или массив, то для этого придется больше потрудиться. Допустим, что требуется предоставить метод `repeat()`, чтобы в результате вызова `Arrays.repeat(n, obj)` был создан массив, содержащий `n` копий объекта `obj`. Разумеется, тип элементов такого массива должен быть таким же, как и тип объекта `obj`. Следующая попытка написать метод `repeat()` не пройдет:

```
public static <T> T[] repeat(int n, T obj) {
  T[] result = new T[n]; // ОШИБКА: построить массив new T[...] нельзя!
  for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

В качестве выхода из этого положения у вызывающего кода можно запросить конструктор массивов в виде ссылки на метод, как выделено ниже полужирным.

```
String[] greetings = Arrays.repeat(10, "Hi", String[]::new);
```

Ниже приведена реализация данного метода.

```
public static <T> T[] repeat(int n, T obj, IntFunction<T[]> constr) {
  T[] result = constr.apply(n);
  for (int i = 0; i < n; i++) result[i] = obj;
  return result;
}
```

С другой стороны, у вызывающего кода можно запросить объект класса и воспользоваться рефлексией, как показано ниже.

```
public static <T> T[] repeat(int n, T obj, Class<T> cl) {
  @SuppressWarnings("unchecked") T[] result =
      (T[]) java.lang.reflect.Array.newInstance(cl, n);
  for (int i = 0; i < n; i++) result[i] = obj;
  return result;
}
```

Этот метод вызывается следующим образом:

```
String[] greetings = Arrays.repeat(10, "Hi", String.class);
```

Еще одна возможность состоит в том, чтобы запросить у вызывающего кода выделение массива. Как правило, вызывающему коду разрешается предоставлять массив любой длины — даже нулевой. Если же предоставляемый массив оказывается слишком коротким, то новый массив создается в вызываемом методе с помощью рефлексии, как показано ниже.

```
public static <T> T[] repeat(int n, T obj, T[] array) {
  T[] result;
  if (array.length >= n)
    result = array;
  else {
    @SuppressWarnings("unchecked") T[] newArray =
        (T[]) java.lang.reflect.Array.newInstance(
            array.getClass().getComponentType(), n);
    result = newArray;
  }
  for (int i = 0; i < n; i++) result[i] = obj;
  return result;
}
```

СОВЕТ. С помощью обобщенного типа все-таки можно получить экземпляр класса **ArrayList**. Так, следующий фрагмент кода вполне допустим:

```
public static <T> ArrayList<T> repeat(int n, T obj) {
  ArrayList<T> result = new ArrayList<>(); // ВЕРНО!
  for (int i = 0; i < n; i++) result.add(obj);
  return result;
}
```

Это намного более простой способ, чем приведенные выше обходные приемы. Поэтому пользоваться им рекомендуется всякий раз, когда нет побудительных причин для построения массива.

НА ЗАМЕТКУ. Если в обобщенном классе требуется обобщенный массив, относящийся к скрытой части реализации, то можно обойтись и построением массива типа `Object[]`. Именно так и делается в классе **`ArrayList`**, как показано ниже.

```
public class ArrayList<E> {
  private Object[] elementData;

  public E get(int index) {
    return (E) elementData[index];
  }
  ...
}
```

6.6.4. Нельзя построить массивы параметризированных типов

Допустим, что требуется создать массив объектов типа `Entry` следующим образом:

```
Entry<String, Integer>[] entries = new Entry<String, Integer>[100];
// ОШИБКА: построить массив из элементов обобщенного типа нельзя!
```

Приведенная выше строка кода содержит синтаксическую ошибку. Такое построение массива запрещено, потому что после стирания типов конструктор массива создаст массив базового типа `Entry`. И тогда станет возможным ввести объекты типа `Entry` любого типа (например, `Entry<Employee, Manager>`), не приводя к исключению типа `ArrayStoreException`.

Следует иметь в виду, что *тип* `Entry<String, Integer>[]` совершенно допустим. В частности, можно объявить переменную данного типа. И если потребуется инициализировать ее, это можно будет сделать следующим образом:

```
@SuppressWarnings("unchecked") Entry<String, Integer>[] entries =
    (Entry<String, Integer>[]) new Entry<?, ?>[100];
```

Но проще воспользоваться списочным массивом таким образом:

```
ArrayList<Entry<String, Integer>> entries = new ArrayList<>(100);
```

Напомним, что параметр переменной длины является скрытым массивом. Если же такой параметр оказывается обобщенным, то ограничение, накладываемое на создание обобщенных массивов, можно обойти. В качестве примера рассмотрим следующий метод:

```
public static <T> ArrayList<T> asList(T... elements) {
  ArrayList<T> result = new ArrayList<>();
```

```
  for (T e : elements) result.add(e);
  return result;
}
```

А теперь рассмотрим следующий вызов:

```
Entry<String, Integer> entry1 = ...;
Entry<String, Integer> entry2 = ...;
ArrayList<Entry<String, Integer>> entries =
          Lists.asList(entry1, entry2);
```

Тип, выводимый из обобщенного типа **T**, также является обобщенным типом `Entry<String, Integer>`, и поэтому массив `elements` относится к типу `Entry<String, Integer>`. Такого рода создания массивов нельзя добиться самостоятельно!

В данном случае компилятор выдает предупреждение, а не ошибку. Если метод только читает элементы из массива параметров, то в его объявлении следует указать аннотацию `@SafeVarargs`, чтобы подавить предупреждение:

```
@SafeVarargs public static <T> ArrayList<T> asList(T... elements)
```

6.6.5. Нельзя употреблять параметры типа класса в статическом контексте

Рассмотрим в качестве примера класс `Entry<K, V>` с параметрами типа. Его параметры типа **K** и **V** нельзя употреблять вместе со статическими переменными или методами. Так, приведенный ниже фрагмент кода оказывается неработоспособным. Ведь стирание типов означает, что в стираемом классе `Entry` имеется лишь одна такая переменная или метод, а не по одному из них для каждого параметра типа **K** и **V**.

```
public class Entry<K, V> {
  private static V defaultValue;
    // ОШИБКА: тип V употребляется в статическом контексте!
  public static void setDefault(V value) { defaultValue = value; }
    // ОШИБКА: тип V употребляется в статическом контексте!
  ...
}
```

6.6.6. Методы не должны конфликтовать после стирания

Нельзя объявлять методы, способные вызвать конфликт после стирания типов. Например, следующий фрагмент кода содержит ошибку:

```
public interface Ordered<T> extends Comparable<T> {
  public default boolean equals(T value) {
    // ОШИБКА: стирание типов приводит к конфликту
    // с методом Object.equals()!
    return compareTo(value) == 0;
  }
  ...
}
```

Метод `equals(T value)` стирается в метод `equals(Object value)`, который вступает в конфликт с одноименным методом из класса `Object`. Но иногда причина

возникающего конфликта не столь очевидна. Ниже приведен пример подобной не очень приятной ситуации.

```
public class Employee implements Comparable<Employee> {
  ...
  public int compareTo(Employee other) {
    return name.compareTo(other.name);
  }
}

public class Manager extends Employee implements Comparable<Manager> {
  // ОШИБКА: две реализации интерфейса Comparable
  // в качестве супертипов не разрешаются!
  ...
  public int compareTo(Manager other) {
    return Double.compare(salary, other.salary);
  }
}
```

Класс `Manager` расширяет класс `Employee` и поэтому воспринимает интерфейс `Comparable<Employee>` как свой супертип. Естественно, что руководителей нужно сравнивать по зарплате, а не по имени. Но почему бы и нет? Здесь нет никакого стирания типов, а только имеются два следующих метода:

```
public int compareTo(Employee other)
public int compareTo(Manager other)
```

Но дело в том, что мостовые методы *вступают в конфликт*. Как пояснялось ранее в разделе 6.5.3, оба эти метода синтезируют следующий мостовой метод:

```
public int compareTo(Object other)
```

6.6.7. Исключения и обобщения

Объекты обобщенного класса нельзя генерировать или перехватывать. В действительности нельзя даже сформировать обобщенный подкласс, расширяющий интерфейс `Throwable`:

```
public class Problem<T> extends Exception
// ОШИБКА: обобщенный класс не может быть подтипом,
// расширяющим интерфейс Throwable!
```

А переменную или параметр типа нельзя употреблять в операторе `catch`, как следует из приведенного ниже примера.

```
public static <T extends Throwable> void doWork(Runnable r, Class<T> cl) {
  try {
    r.run();
  } catch (T ex) { // ОШИБКА: перехватить переменную типа нельзя!
    Logger.getGlobal().log(..., ..., ex);
  }
}
```

Тем не менее переменную или параметр типа *можно* указывать в объявлении с оператором `throws` следующим образом:

```
public static <V, T> V doWork(Callable<V> c, T ex) throws T {
  try {
    return c.call();
  } catch (Throwable realEx) {
    ex.initCause(realEx);
    throw ex;
  }
}
```

ВНИМАНИЕ. С помощью обобщений можно устранить различие между проверяемыми и непроверяемыми исключениями. Для этой цели служит следующая пара методов:

```
public class Exceptions {
  @SuppressWarnings("unchecked")
  private static <T extends Throwable>
          void throwAs(Throwable e) throws T {
    throw (T) e; // Приведение обобщенного типа
                 // стирается в тип (Throwable) e
  }
  public static <V> V doWork(Callable<V> c) {
    try {
      return c.call();
    } catch (Throwable ex) {
      Exceptions.<RuntimeException>throwAs(ex);
      return null;
    }
  }
}
```

А теперь рассмотрим следующий метод:

```
public static String readAll(Path path) {
  return doWork(() -> new String(Files.readAllBytes(path)));
}
```

Несмотря на то что при вызове метода **`Files.readAllBytes()`** генерируется проверяемое исключение, если путь к файлу не найден, это исключение не объявляется и не перехватывается в методе **`readAll()`**!

6.7. Обобщения и рефлексия

В последующих разделах будет показано, что можно сделать с обобщенными классами в пакете рефлексии и как выявить в виртуальной машине небольшое количество информации об обобщенном типе, сохраняющейся в процессе стирания.

6.7.1. Класс `Class<T>`

У класса `Class` имеется параметр типа, обозначающий тот класс, который описывается объектом типа `Class`. Разъясним это замысловатое утверждение по порядку.

В качестве примера рассмотрим класс `String`. В виртуальной машине имеется объект типа `Class`, который можно получить, сделав вызов `"Fred".getClass()`, или непосредственно в виде литерала класса `String.class`. С помощью этого объекта можно выяснить, какие методы имеются в классе `String` и как получить его экземпляр.

В последнем случае оказывает помощь параметр типа. Метод `newInstance()` объявляется следующим образом:

```
public Class<T> {
  public T newInstance() throws ... { ... }
}
```

Таким образом, этот метод возвращает объект типа `T`. А литерал класса `String.class` относится к типу `Class<String>` потому, что его метод `newInstance()` возвращает объект типа `String`.

Эти сведения о классе позволяют сэкономить на приведении типов. В качестве примера рассмотрим следующий метод:

```
public static <T> ArrayList<T> repeat(int n, Class<T> cl)
       throws ReflectiveOperationException {
  ArrayList<T> result = new ArrayList<>();
  for (int i = 0; i < n; i++) result.add(cl.newInstance());
  return result;
}
```

Этот метод компилируется, поскольку в результате вызова `cl.newInstance()` возвращается обобщенный тип `T`. Допустим, что этот метод вызывается следующим образом: `repeat(10, Employee.class)`. В таком случае из обобщенного типа `T` автоматически выводится тип `Employee`, поскольку литерал класса `Employee.class` относится к типу `Class<Employee>`. Следовательно, возвращаемым оказывается тип `ArrayList<Employee>`.

Помимо метода `newInstance()`, в классе `Class` имеется ряд других методов, в которых употребляется параметр типа. К их числу относятся следующие:

```
Class<? super T> getSuperclass()
<U> Class<? extends U> asSubclass(Class<U> clazz)
T cast(Object obj)
Constructor<T> getConstructor(Class<?>... parameterTypes)
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
T[] getEnumConstants()
```

Как было показано в главе 4, имеется немало случаев, когда ничего неизвестно о классе, описываемом объектом типа `Class`. В подобных случаях можно просто воспользоваться подстановочным типом `Class<?>`.

6.7.2. Сведения об обобщенном типе в виртуальной машине

Стирание оказывает воздействие только на параметры конкретизируемого типа. Полные сведения об *объявлении* обобщенных классов и методов доступны во время выполнения.

Допустим, что в результате вызова obj.getClass() возвращается литерал класса ArrayList.class. Трудно сказать, был ли объект obj построен как экземпляр класса ArrayList<String> или ArrayList<Employee>. Но можно сказать, что класс ArrayList является обобщенным с параметром типа E, не имеющим никаких ограничений.

Аналогично рассмотрим следующий метод из класса Collections:

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Соответствующий объект типа Method можно получить, как пояснялось в главе 4 и показано ниже. Из этого объекта типа Method можно восстановить всю сигнатуру метода.

```
Method m = Collections.class.getMethod("sort", List.class);
```

Интерфейс Type из пакета java.lang.reflect представляет объявления обобщенных типов. У этого интерфейса имеются следующие подтипы.

1. Класс Class, описывающий конкретные типы.
2. Интерфейс TypeVariable, описывающий переменные и параметры типа (например, T extends Comparable<? super T>).
3. Интерфейс WildcardType, описывающий подстановочные типы (например, ? super T).
4. Интерфейс ParameterizedType, описывающий типы обобщенных классов или интерфейсов (например, Comparable<? super T>).
5. Интерфейс GenericArrayType, описывающий обобщенные массивы (например, T[]).

Следует иметь в виду, что четыре последних подтипа из перечисленных выше являются интерфейсами. Виртуальная машина конкретизирует подходящие классы, реализующие эти интерфейсы.

Классы и методы имеют переменные и параметры типа. Формально конструкторы не являются методами и представлены отдельным классом в библиотеке рефлексии. Они также могут быть обобщенными. Чтобы выяснить, происходит ли объект типа Class, Method или Constructor от обобщенного объявления, следует вызвать метод getTypeParameters(). В итоге получается массив экземпляров типа TypeVariable (по одному на каждую переменную или параметр типа в объявлении) или же массив нулевой длины, если объявление не было обобщенным.

Интерфейс TypeVariable<D> является обобщенным. Его параметр типа относится к типу Class<T>, Method или Constructor<T> в зависимости от места его объявления. Так, в следующем примере кода демонстрируется, как получить параметр типа для класса ArrayList:

```
TypeVariable<Class<ArrayList>>[] vars = ArrayList.class.getTypeParameters();
String name = vars[0].getName(); // "E"
```

А в следующем примере кода получается переменная типа для метода Collections.sort():

```
Method m = Collections.class.getMethod("sort", List.class);
TypeVariable<Method>[] vars = m.getTypeParameters();
String name = vars[0].getName(); // "T"
```

На эту переменную накладывается ограничение, которое можно обработать следующим образом:

```
Type[] bounds = vars[0].getBounds();
if (bounds[0] instanceof ParameterizedType) { // Comparable<? super T>
  ParameterizedType p = (ParameterizedType) bounds[0];
  Type[] typeArguments = p.getActualTypeArguments();
  if (typeArguments[0] instanceof WildcardType) { // ? super T
    WildcardType t = (WildCardType) typeArguments[0];
    Type[] upper = t.getUpperBounds(); // ? extends ... & ...
    Type[] lower = t.getLowerBounds(); // ? super ... & ...
    if (lower.length > 0) {
      String description = lower[0].getTypeName(); // "T"
      ...
    }
  }
}
```

Приведенные выше примеры дают некоторое представление о том, как можно анализировать обобщенные объявления. Эти примеры не рассматриваются подробно, поскольку они не характерны для обычной практики программирования на Java. Самое главное, что объявления обобщенных классов и методов не стираются и доступны посредством рефлексии.

Упражнения

1. Реализуйте обобщенный класс `Stack<E>`, управляющий списочным массивом, состоящим из элементов типа **E**. Предоставьте методы `push()`, `pop()` и `isEmpty()`.
2. Еще раз реализуйте обобщенный класс `Stack<E>`, используя массив для хранения элементов. Если требуется, нарастите массив в методе `push()`. Предоставьте два решения этой задачи: одно — с массивом типа `E[]`, другое — с массивом типа `Object[]`. Оба решения должны компилироваться без всяких предупреждений. Какое из них вы предпочтете сами и почему?
3. Реализуйте обобщенный класс `Table<K, V>`, управляющий списочным массивом, состоящим из элементов типа `Entry<K, V>`. Предоставьте методы для получения значения, связанного с ключом, установки значения по заданному ключу и удаления ключа.
4. Сделайте вложенным класс `Entry` из предыдущего упражнения. Должен ли этот класс быть обобщенным?
5. Рассмотрите следующий вариант метода `swap()`, где массив может быть предоставлен с помощью аргументов переменной длины:

```
public static <T> T[] swap(int i, int j, T... values) {
  T temp = values[i];
```

```
    values[i] = values[j];
    values[j] = temp;
    return values;
}
```

А теперь рассмотрите следующий вызов:

```
Double[] result = Arrays.swap(0, 1, 1.5, 2, 3);
```

Какое сообщение об ошибке вы получите? Далее сделайте такой вызов:

```
Double[] result = Arrays.<Double>swap(0, 1, 1.5, 2, 3);
```

Изменилось ли к лучшему сообщение об ошибке? Что нужно сделать для устранения ошибки?

6. Реализуйте обобщенный метод, присоединяющий все элементы из одного списочного массива к другому. Воспользуйтесь метасимволом подстановки для обозначения одного из аргументов типа. Предоставьте два равнозначных решения: одно с подстановочным типом `? extends E`, другое — с подстановочным типом `? super E`.

7. Реализуйте обобщенный класс `Pair<E>`, позволяющий сохранять пару элементов типа `E`. Предоставьте методы доступа для получения первого и второго элементов.

8. Видоизмените класс из предыдущего упражнения, введя методы `max()` и `min()` для получения наибольшего и наименьшего из двух элементов. Наложите соответствующее ограничение на обобщенный тип `E`.

9. Предоставьте в служебном классе `Arrays` следующий метод, возвращающий пару, состоящую из первого и последнего элементов массива `a`, указав подходящий аргумент типа:

```
public static <E> Pair<E> firstLast(ArrayList<___> a)
```

10. Предоставьте в служебном классе `Arrays` обобщенные методы `min()` и `max()`, возвращающие наименьший и наибольший элементы массива соответственно.

11. Продолжая предыдущее упражнение, предоставьте метод `minMax()`, возвращающий объект типа `Pair` с наименьшим и наибольшим элементами массива.

12. Реализуйте следующий метод, сохраняющий наименьший и наибольший элементы из массива `elements` в списке `result`:

```
public static <T> void minmax(List<T> elements,
       Comparator<? super T> comp, List<? super T> result)
```

Обратите внимание на подстановочный тип в последнем параметре. Для хранения полученного результата подойдет любой супертип обобщенного типа `T`.

13. С учетом метода из предыдущего упражнения рассмотрите следующий метод:

```
public static <T> void maxmin(List<T> elements,
       Comparator<? super T> comp, List<? super T> result) {
  minmax(elements, comp, result);
```

```
  Lists.swapHelper(result, 0, 1);
}
```

Почему этот метод нельзя скомпилировать без захвата подстановки? *Подсказка:* попробуйте предоставить явный тип `Lists.<___>swapHelper(result, 0, 1)`.

14. Реализуйте усовершенствованный вариант метода `closeAll()`, представленного в разделе 6.3. Закройте все элементы даже в том случае, если некоторые из них генерируют исключение. В таком случае сгенерируйте исключение впоследствии. Если исключение генерируется в результате двух или больше вызовов данного метода, свяжите их в цепочку.

15. Реализуйте метод `map()`, получающий списочный массив и объект типа `Function<T, R>` (см. главу 3) и возвращающий списочный массив, состоящий из результатов применения функции к заданным элементам этого массива.

16. К чему приведет стирание типов в приведенных ниже методах из класса `Collection`?

```
public static <T extends Comparable<? super T>>
    void sort(List<T> list)
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

17. Определите класс `Employee`, реализующий интерфейс `Comparable<Employee>`. Используя утилиту **`javap`**, продемонстрируйте, что мостовой метод был синтезирован. Каково его назначение?

18. Рассмотрите следующий метод, представленный в разделе 6.6.3:

```
public static <T> T[] repeat(int n, T obj, IntFunction<T[]> constr)
```

Почему исход вызова `Arrays.repeat(10, 42, int[]::new)` окажется неудачным? Как устранить этот недостаток? Что нужно сделать для применения других примитивных типов?

19. Рассмотрите следующий метод, представленный в разделе 6.6.3:

```
public static <T> ArrayList<T> repeat(int n, T obj)
```

Этот метод без особых хлопот составляет списочный массив типа `ArrayList<T>` из элементов обобщенного типа **`T`**. Можно ли получить массив типа `T[]` из этого списочного массива, не пользуясь объектом типа `Class` или ссылкой на конструктор? Если нельзя, то почему?

20. Реализуйте следующий метод:

```
@SafeVarargs public static final <T> T[] repeat(int n, T... objs)
```

Он должен возвращать массив `n` копий заданных объектов. Имейте в виду, что для этого не потребуется объект типа `Class` или ссылка на конструктор, поскольку наращивать количество объектов `objs` можно рефлексивно.

21. Используя аннотацию `@SafeVarargs`, напишите метод, позволяющий строить массивы обобщенных типов, как в следующем примере:

```
List<String>[] result = Arrays.<List<String>>construct(10);
  // Устанавливает результат в списке типа List<String>[] длиной 10
```

22. Усовершенствуйте метод `public static <V, T> V doWork(Callable<V> c, T ex) throws T`, представленный в разделе 6.6.7, таким образом, чтобы передавать ему объект исключения, который вряд ли будет вообще использован. Вместо этого данный метод должен принимать ссылку на класс исключения.

23. Во врезке "Внимание!" из раздела 6.6.7 упоминается вспомогательный метод `throwAs()`, применяемый для "приведения" типа исключения `ex` к типу `RuntimeException` и его генерирования. Почему для этой цели нельзя воспользоваться обычным приведением типов, т.е. `throw (RuntimeException) ex`?

24. Какие методы можно вызвать для переменной типа `Class<?>`, не прибегая к приведению типов?

25. Напишите метод `public static String genericDeclaration(Method m)`, возвращающий объявление метода `m()`, перечисляющего параметры типа с их ограничениями и типами параметров метода, включая их аргументы типа, если это обобщенные типы.

Глава 7

Коллекции

В этой главе...

Для программистов разработано немало структур данных, где они могут эффективно хранить значения и извлекать их. В прикладном программном интерфейсе Java API предоставляются реализации наиболее употребительных структур данных и алгоритмов обращения с ними, а также специальный каркас для их организации. В этой главе поясняется, как обращаться со списками, множествами, отображениями и прочими разновидностями коллекций.

Основные положения этой главы приведены ниже.

1. Интерфейс `Collection` предоставляет общие методы для всех коллекций, кроме отображений, которые описывает интерфейс `Map`.
2. Список представляет собой последовательную коллекцию, где каждый элемент имеет целочисленный индекс.
3. Множество оптимизировано для эффективной проверки вложенности. В языке Java предоставляются реализации множеств в виде классов `HashSet` и `TreeSet`.
4. Отображения реализованы в виде классов `HashMap` и `TreeMap`. А отображение типа `LinkedHashMap` сохраняет порядок ввода элементов.
5. Интерфейс `Collection` и класс `Collections` предоставляют немало полезных алгоритмов для операций над множествами, поиска, сортировки, перетасовки и пр.
6. Представления обеспечивают доступ к данным, хранящимся в другом месте, используя стандартные интерфейсы коллекций.

7.1. Краткий обзор каркаса коллекций

Каркас коллекций в Java предоставляет реализации общих структур данных. Чтобы упростить написание прикладного кода, зависящего от выбора структур данных, в каркасе коллекций предоставляется целый ряд общих интерфейсов (рис. 7.1). Основным среди них считается интерфейс `Collection`, методы которого перечислены в табл. 7.1.

Таблица 7.1. Методы из интерфейса **`Collection<E>`**

Метод	Описание
`boolean add(E e)`	Вводят заданную коллекцию ***e*** или элементы из заданной коллекции ***c***
`boolean addAll(Collection<? extends E> c)`	Возвращают логическое значение **`true`**, если коллекция изменилась
`boolean remove(Object o)`	Удаляют заданную коллекцию ***o***, элементы из заданной коллекции ***c***,
`boolean removeAll(Collection<?> c)`	элементы, отсутствующие в заданной коллекции ***c***, совпадающие или все элементы
`boolean retainAll(Collection<?> c)`	Первые четыре метода возвращают логическое значение **`true`**, если коллекция изменилась
`boolean removeIf(Predicate<? super E> filter)`	

Окончание табл. 7.1

Метод	Описание
`void clear()`	
`int size()`	Возвращает количество элементов в данной коллекции
`boolean isEmpty()`	Возвращают логическое значение `true`, если коллекция пуста,
`boolean contains(Object o)`	содержит заданную коллекцию *o*
`boolean containsAll(Collection<?> c)`	или все элементы заданной коллекции *c*
`Iterator<E> iterator()`	Возвращают итератор, поток данных, а возможно, и параллельный поток данных или итератор-разделитель для перебора элементов данной коллекции
`Stream<E> stream()`	Подробнее об итераторах см. в разделе 7.2, а о потоках данных — в главе 8
`Stream<E> parallelStream()`	Итераторы-разделители представляют интерес только для разработчиков потоков данных
`Spliterator<E> spliterator()`	
`Object[] toArray()`	Возвращает массив, состоящий из элементов данной коллекции
`T[] toArray(T[] a)`	Возвращает массив a, состоящий из элементов данной коллекции, если у него достаточная длина

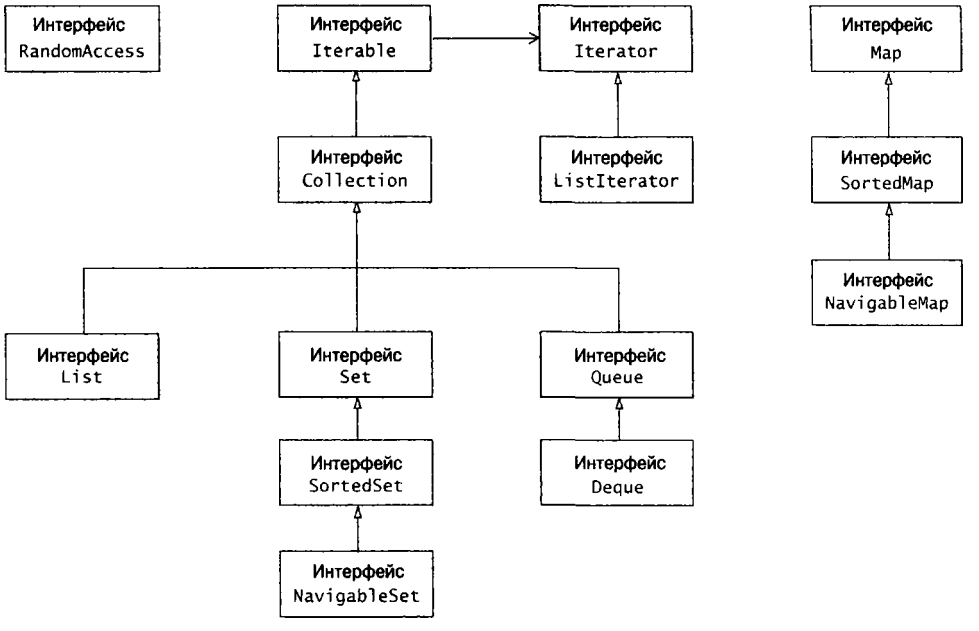


Рис. 7.1. Интерфейсы из каркаса коллекций в Java

Интерфейс List представляет последовательную коллекцию, состоящую из элементов на позициях 0, 1, 2 и т.д. В табл. 7.2 перечислены методы из этого интерфейса.

Таблица 7.2. Методы из интерфейса List

Метод	Описание
`boolean add(int index, E e)`	Вводят заданную коллекцию *e* или элементы из заданной коллекции *c*, до указанной позиции *index* или до конца коллекции
`boolean add(int index, Collection<? extends E> c)`	Возвращают логическое значение `true`, если список изменился
`boolean add(E e)`	
`boolean add(Collection <? extends E> c)`	
`E get(int index)`	Получают, устанавливают или удаляют элемент из списка по указанному индексу
`E set(int index, E element)`	Два последних метода возвращают элемент, находившийся в списке по указанному индексу перед их вызовом
`E remove(int index)`	
`int indexOf(Object o)`	Возвращают индекс первого или последнего элемента в списке,
`int lastIndexOf(Object o)`	равный 0 или `-1`, если отсутствует совпадение
`ListIterator<E> listIterator()`	Возвращают список итераторов для всех элементов или части элементов,
`ListIterator<E> listIterator(int index)`	начиная с указанной позиции *index*
`void replaceAll(Unary Operator<E> operator)`	Заменяет каждый элемент результатом выполнения над ним указанной операции
`void sort(Comparator <? super E> c)`	Сортирует этот список, используя заданное упорядочение *c*
`List<E> subList(int fromIndex, int toIndex)`	Возвращает представление (см. раздел 7.6) подсписка, начиная с заданной позиции *fromIndex* и до позиции, предшествующей заданной позиции *toIndex*

Интерфейс List реализуется классом ArrayList, представленным ранее в данной книге, а также классом LinkedList. Если вы когда-нибудь изучали структуры данных, то, вероятно, помните, что связный список — это последовательность связанных вместе узлов, каждый из которых содержит элемент данного списка. Чтобы быстро ввести элемент посредине связного списка, достаточно нарастить узел. Но для того чтобы добраться до средины связного списка, придется отследить все связи с самого его начала, что замедляет дело. Связные списки, конечно, находят свое применение, но большинство прикладных программистов предпочитают списочные массивы, если им требуется последовательная коллекция. Тем не менее интерфейс List полезен. Например, метод `Collections.nCopies(n, o)` возвращает объект типа List с n копиями объекта o. Этот объект вводит в заблуждение тем, что он на самом деле не хранит n копий, но когда запрашивается любая из них, возвращает объект o.

ВНИМАНИЕ. Интерфейс **List** предоставляет методы для доступа к *n*-му элементу списка, даже если такой доступ может оказаться неэффективным. Чтобы указать на наличие такого доступа, класс коллекции должен реализовать маркерный интерфейс **RandomAccess** без методов. Например, класс **ArrayList** реализует интерфейсы **List** и **RandomAccess**, тогда как класс **LinkedList** — только интерфейс **List**.

В множество, представленное интерфейсом Set, элементы не вводятся на конкретной позиции, а дублирующиеся элементы не допускаются. Множество, представленное интерфейсом SortedSet, допускает итерацию своих элементов по порядку сортировки, а в интерфейсе NavigableSet имеются методы для поиска соседних элементов в множестве. Подробнее о множествах речь пойдет в разделе 7.3.

Очередь, представленная интерфейсом Queue, сохраняет порядок ввода в нее элементов, но их можно вводить только в хвост очереди, а удалять — из ее головы (подобно живой очереди). Очередь, представленная интерфейсом Deque, имеет двусторонний доступ, т.е. элементы вводятся и удаляются из нее с обоих ее концов.

Интерфейсы всех коллекций являются обобщенными, причем параметр типа обозначает тип элементов коллекции (например, Collection<E>, List<E> и т.д.). В интерфейсе Map<K, V> параметр типа K обозначает тип ключа, а параметр типа V — тип значения.

Рекомендуется как можно больше пользоваться интерфейсами в прикладном коде. Например, после построения списочного массива типа ArrayList ссылку на него нужно сохранить в переменной типа List следующим образом:

```
List<String> words = new ArrayList<>();
```

Всякий раз, кода реализуется метод, обрабатывающий коллекцию, в качестве параметра типа следует выбирать наименее ограничительный интерфейс. Для этой цели лучше всего подходит интерфейс Collection, List или Map.

Одно из преимуществ каркаса коллекций заключается в том, что он избавляет от необходимости изобретать колесо, предоставляя общие алгоритмы обращения с коллекциями. Некоторые из основных алгоритмов реализованы в методах (например, addAll() и removeIf()) из интерфейса Collection. Служебный класс Collections предоставляет немало дополнительных алгоритмов для обращения с разными видами коллекций. Они позволяют сортировать, тасовать, циклически сдвигать и обращать списки, находить максимальное и минимальное значения или позицию произвольного элемента в коллекции и формировать коллекции без элементов, с одним элементом или n копиями одного и того же элемента. Наиболее употребительные методы из класса Collections сведены в табл. 7.3.

Таблица. 7.3. Полезные методы из класса **Collections**

Метод (все методы статические)	Описание
boolean disjoint(Collection<?> *c1*, Collection<?> *c2*)	Возвращает логическое значение **true**, если в сравниваемых коллекциях отсутствуют общие элементы
boolean addAll(Collection<? super T> *c*, T... *elements*)	Вводит все элементы в заданную коллекцию *c*

Продолжение табл. 7.3

Метод (все методы статические)	Описание
`void copy(List<? super T> dest, List<? extends T> src)`	Копирует все элементы из исходной коллекции *src* в целевую коллекцию *dest*, которая должна быть не меньше исходной, по тем же самым индексам
`boolean replaceAll(List<T> list, T oldVal, T newVal)`	Заменяет все прежние элементы *oldVal* новыми элементами *newVal*, каждый из которых может быть пустым (`null`). Возвращает логическое значение `true`, если найдено хотя бы одно совпадение
`void fill(List<? super T> list, T obj)`	Устанавливает указанный объект *obj* во всех элементах списка
`List<T> nCopies(int n, T o)`	Получает неизменяемый список, состоящий из *n* копий указанного объекта *o*
`int frequency(Collection<?> c, Object o)`	Возвращает количество элементов из заданной коллекции, равных указанному объекту *o*
`int indexOfSubList(List<?> source, List<?> target)`	Возвращают начало первого или последнего вхождения целевого списка в исходный список,
`int lastIndexOfSubList(List<?> source, List<?> target)`	а в их отсутствие — значение `-1`
`int binarySearch(List<? extends Comparable<? super T>> list, T key)`	Возвращают позицию ключа, при условии, что список отсортирован в естественном порядке следования элементов или заданном порядке *c*
`int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`	Если ключ отсутствует, возвращается разность `-i - 1`, где `i` — позиция, на которой должен быть введен ключ
`sort(List<T> list)`	Сортируют заданный список в естественном порядке следования элементов
`sort(List<T> list, Comparator<? super T> c)`	или заданном порядке *c*
`void swap(List<?> list, int i, int j)`	Переставляет элементы заданного списка на указанных позициях
`void rotate(List<?> list, int distance)`	Выполняет циклический сдвиг списка, перемещая элемент с позиции по индексу *i* на позицию по индексу `(i + distance) % list.size()`
`void reverse(List<?> list)`	Обращают или
`void shuffle(List<?> list)`	произвольно перетасовывают
`void shuffle(List<?> list, Random rnd)`	заданный список
`Set<T> singleton(T o)`	Получают одноэлементное множество,
`List<T> singletonList(T o)`	список или отображение
`Map<K, V> singletonMap(K key, V value)`	
`empty(List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap)()`	Получает пустое представление (см. далее раздел 7.6)
`synchronized(Collection|List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap)()`	Получает синхронизированное представление (см. далее раздел 7.6)

Окончание табл. 7.3

Метод (все методы статические)	Описание
`unmodifiable(Collection|List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap)()`	Получает неизменяемое представление (см. далее раздел 7.6)
`checked(Collection|List|Set|SortedSet|NavigableSet|Map|SortedMap|NavigableMap|Queue)()`	Получает проверяемое представление (см. далее раздел 7.6)

7.2. Итераторы

Каждая коллекция предоставляет свой способ перебора ее элементов в определенном порядке. Для этой цели в интерфейсе `Iterable<T>`, служащем супертипом для интерфейса `Collection`, определяется следующий метод:

```
Iterator<T> iterator()
```

Этот метод получает итератор, которым можно пользоваться для обхода всех элементов коллекции, как показано ниже.

```
Collection<String> coll = ...;
Iterator<String> iter = coll.iterator();
  while (iter.hasNext()) {
    String element = iter.next();
    Обработать элемент коллекции element
}
```

В данном случае достаточно воспользоваться расширенным циклом `for` следующим образом:

```
for (String element : coll) {
  Обработать элемент коллекции element
}
```

НА ЗАМЕТКУ. Для *любого* объекта **c** класса, реализующего интерфейс **`Iterable<E>`**, расширенный цикл **`for`** преобразуется в приведенную выше форму.

У интерфейса `Iterator` имеется также метод `remove()`, удаляющий перебранный ранее элемент. Так, в следующем цикле из коллекции удаляются все элементы, удовлетворяющие заданному условию:

```
while (iter.hasNext()) {
  String element = iter.next();
  if (element удовлетворяет заданному условию)
    iter.remove();
}
```

Но для этой цели проще воспользоваться методом `removeIf()` следующим образом:

```
coll.removeIf(e -> e удовлетворяет заданному условию);
```

ВНИМАНИЕ. Метод `remove()` удаляет из коллекции последний элемент, возвращаемый итератором, а не тот элемент, на который указывает итератор. Поэтому вызов метода `remove()` следует непременно перемежать с вызовом метода `next()` или `previous()`, а не вызывать его два раза подряд.

Интерфейс `ListIterator` является подчиненным интерфейсу `Iterator` и содержит методы для ввода элемента перед итератором, установки другого значения в перебираемом элементе и перемещения в коллекции назад. Этот интерфейс наиболее удобен для обращения со связными списками.

```
List<String> friends = new LinkedList<>();
ListIterator<String> iter = friends.listIterator();
iter.add("Fred"); // Fred |
iter.add("Wilma"); // Fred Wilma |
iter.previous(); // Fred | Wilma
iter.set("Barney"); // Fred | Barney
```

ВНИМАНИЕ. Если для обхода структуры данных имеется несколько итераторов и один из них изменяет ее, то другой может стать недействительным. Если же и дальше пользоваться недействительным итератором, он может сгенерировать исключение типа `ConcurrentModificationException`.

7.3. Множества

Множество, по существу, проверяет, является ли указанное значение его элементом, но оно дает и кое-что взамен, не запоминая порядок, в котором были введены элементы. Множества удобны в тех случаях, когда порядок расположения элементов не имеет значения. Так, если требуется множество неприличных слов, запрещенных в качестве имени пользователя, то порядок их расположения в множестве особого значения не имеет. Нужно лишь выяснить, находится предлагаемое имя пользователя в этом множестве.

Интерфейс `Set`, представляющий множество, реализуется классами `HashSet` и `TreeSet`. Но внутренняя реализация данного интерфейса в обоих классах заметно отличается. Если вам приходилось прежде изучать структуры данных, то вы, вероятно, знаете, как реализуются хеш-таблицы и двоичные деревья. Впрочем, пользоваться классами `HashSet` и `TreeSet` можно, даже не зная их внутренней реализации.

Как правило, хеш-множество оказывается более эффективным, при условии, что имеется подходящая *хеш-функция* для его элементов. Такие библиотечные классы, как `String` или `Path`, имеют подходящие хеш-функции. О том, как писать подходящие хеш-функции для своих классов, см. в главе 4.

Например, упомянутое выше множество неприличных слов может быть реализовано следующим образом:

```
Set<String> badWords = new HashSet<>();
badWords.add("sex");
badWords.add("drugs");
badWords.add("c++");
```

```
if (badWords.contains(username.toLowerCase()))
  System.out.println("Please choose a different user name");
```

Если множество требуется обойти в отсортированном порядке, то для этой цели следует воспользоваться классом `TreeSet`. Одной из причин для этого может служить предоставление пользователям отсортированного списка вариантов выбора.

Тип элемента множества должен относиться к классу, реализующему интерфейс `Comparable`. В противном случае в конструкторе класса придется указать интерфейс `Comparator`. Ниже приведен пример построения древовидного множества типа `TreeSet`.

```
TreeSet<String> countries = new TreeSet<>(); // Обходит названия стран,
       // введенные в древовидное множество, в отсортированном порядке
countries = new TreeSet<>((u, v) ->
    u.equals(v) ? 0
    : u.equals("USA") ? -1
    : v.equals("USA") ? 1
    : u.compareTo(v));
    // Название "USA" всегда следует первым
```

Класс `TreeSet` реализует интерфейсы `SortedSet` и `NavigableSet`, методы которых перечислены в табл. 7.4 и 7.5 соответственно.

Таблица 7.4. Методы из интерфейса **`SortedSet<E>`**

Метод	Описание
`E first()`	Возвращают первый и последний элементы в множестве
`E last()`	
`SortedSet<E> headSet(E toElement)`	Возвращают представление элементов множества,
`SortedSet<E> subSet(E fromElement, E toElement)`	начиная с заданной позиции ***`fromElement`*** и до позиции,
`SortedSet<E> tailSet(E fromElement)`	предшествующей заданной позиции ***`toElement`***

Таблица 7.5. Методы из интерфейса **`NavigableSet<E>`**

Метод	Описание
`E higher(E e)`	Возвращают из множества ближайший элемент больше,
`E ceiling(E e)`	больше или равный,
`E floor(E e)`	меньше или равный,
`E lower(E e)`	меньше заданного элемента ***`e`***
`E pollFirst()`	Удаляет и возвращает из множества первый или последний элемент,
`E pollLast()`	а если множество оказывается пустым — пустое значение **`null`**

Окончание табл. 7.5

Метод	Описание
NavigableSet<E> headSet(E *toElement*, boolean *inclusive*)	Возвращают представление элементов множества,
NavigableSet<E> subSet(E *fromElement*, boolean *fromInclusive*, E *toElement*, boolean *toExclusive*)	начиная с позиции ***fromElement*** и до позиции ***toElement***
NavigableSet<E> tailSet(E *fromElement*, boolean *inclusive*)	(включительно или исключительно)

7.4. Отображения

В отображениях хранятся связи ключей со значениями. Чтобы ввести новую связь в отображение или изменить в нем значение по существующему ключу, достаточно вызвать метод `put()`:

```
Map<String, Integer> counts = new HashMap<>();
counts.put("Alice", 1); // Вводит пару "ключ-значение" в отображение
counts.put("Alice", 2); // Обновляет значение по заданному ключу
```

В данном примере применяется хеш-отображение, которое представлено классом `HashMap` и, как хеш-множество, больше подходит в том случае, если ключи не требуется обходить в отсортированном порядке. В противном случае следует выбрать древовидное отображение, представленное классом `TreeMap`. Ниже показано, как получить значение, связанное с ключом в отображении:

```
int count = counts.get("Alice");
```

Если ключ отсутствует, метод `get()` возвращает пустое значение `null`. В данном примере это вызовет исключение типа `NullPointerException`, если искомое значение не распаковано. Поэтому лучше воспользоваться приведенным ниже методом, возвращающим подсчет `0`, если ключ отсутствует.

```
int count = counts.getOrDefault("Alice", 0);
```

Если счетчик обновляется в отображении, то сначала нужно проверить его наличие, и если он присутствует, то добавить `1` к существующему значению. Метод `merge()` упрощает эту типичную операцию. Так, в результате вызова

```
counts.merge(word, 1, Integer::sum);
```

ключ `word` связывается со значение `1`, если он прежде отсутствовал, а иначе — к предыдущему значению добавляется `1` с помощью функции `Integer::sum()`.

Методы, реализующие операции с отображением, сведены в табл. 7.6.

Таблица 7.6. Методы из интерфейса **Map<K, V>**

Метод	Описание
V get(Object *key*)	Если заданный ключ ***key*** связан с непустым значением **v**, то возвращает значение **v**,

Продолжение табл. 7.6

Метод	Описание
`V getOrDefault(Object key, V defaultValue)`	а иначе — пустое значение `null` или указанное значение по умолчанию *defaultValue*
`V put(K key, V value)`	Если заданный ключ *key* связан с непустым значением *v*, то связывает ключ *key* с указанным значением *value* и возвращает значение *v*, а иначе вводит запись и возвращает пустое значение `null`
`V putIfAbsent(K key, V value)`	Если заданный ключ *key* связан с непустым значением *v*, то игнорирует указанное значение *value* и возвращает значение *v*, а иначе вводит запись и возвращает пустое значение `null`
`V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)`	Если заданный ключ *key* связан с непустым значением *v*, то применяет указанную функцию к значению *v* и указанному значению *value*, а заданный ключ *key* связывается с полученным результатом или удаляется, если результат пустой (`null`). В противном случае связывает заданный ключ *key* с указанным значением *value*. Возвращает результат вызова `get(key)`
`V compute(K key, BiFunction< ? super K,? super V,? extends V> remappingFunction)`	Применяет указанную функцию к заданному ключу *key* и результату вызова `get(key)`. Заданный ключ *key* связывается с полученным результатом или удаляется, если результат пустой (`null`). Возвращает результат вызова `get(key)`
`V computeIfPresent(K key, BiFunction< ? super K,? super V,? extends V> remappingFunction)`	Если заданный ключ *key* связан с непустым значением *v*, то применяет указанную функцию к заданному ключу *key* и значению *v*. Заданный ключ *key* связывается с полученным результатом или удаляется, если результат пустой (`null`). Возвращает результат вызова `get(key)`
`V computeIfAbsent(K key, Function<? super K,? extends V> remappingFunction)`	Применяет указанную функцию к заданному ключу *key*, если только он не связан с непустым значением *v*. Заданный ключ *key* связывается с полученным результатом или удаляется, если результат пустой (`null`). Возвращает результат вызова `get(key)`
`void putAll(Map<? extends K, ? extends V> m)`	Вводит все записи из указанного отображения *m*
`V remove(Object key)`	Удаляет заданный ключ и связанное с ним значение или заменяет прежнее значение
`V replace(K key, V newValue)`	Возвращают прежнее значение, а если оно отсутствует — пустое значение `null`
`boolean remove(Object key, Object value)`	Удаляют запись из отображения или заменяют прежнее значение и возвращают логическое значение `true`, при условии, что заданный ключ *key* связан с указанным значением *value*. В противном случае ничего не делают и возвращают логическое значение `false`
`boolean replace(K key, V value, V newValue)`	Эти методы представляют интерес в основном для параллельного доступа к отображению
`int size()`	Возвращает количество записей в отображении

Окончание табл. 7.6

Метод	Описание
`boolean isEmpty()`	Проверяет, является ли отображение пустым
`void clear()`	Удаляет все записи из отображения
`void forEach(BiConsumer<? super K, ? super V> action)`	Выполняет заданное действие `action` над всеми записями в отображении
`void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)`	Вызывает указанную функцию для всех записей в отображении. Связывает ключи с непустыми результатами и удаляет ключи с пустыми (`null`) результатами
`boolean containsKey(Object key)`	Проверяют, содержит ли отображение заданный ключ или указанное значение
`boolean containsValue(Object value)`	
`Set<K> keySet()`	Возвращают представления ключей,
`Collection<V> values()`	значения и записи из отображения
`Set<Map.Entry<K, V>> entrySet()`	

Вызвав следующие методы, можно получить *представления* ключей и записей в отображении:

```
Set<K> keySet()
Set<Map.Entry<K, V>> entrySet()
Collection<K> values()
```

Возвращаемые в итоге коллекции не являются копиями данных из отображения, но связаны с отображением. Если удалить ключ или запись из представления, эта запись удаляется также из лежащего в основе отображения.

Чтобы перебрать все ключи и значения в отображении, достаточно перебрать множество, возвращаемое методом `entrySet()`, как показано ниже.

```
for (Map.Entry<String, Integer> entry : counts.entrySet()) {
  String k = entry.getKey();
  Integer v = entry.getValue();
  Обработать пару "ключ-значение"  k, v
}
```

С другой стороны, достаточно вызвать метод `forEach()` следующим образом:

```
counts.forEach((k, v) -> {
  Обработать пару "ключ-значение" k, v
});
```

ВНИМАНИЕ. В некоторых реализациях отображений (например, в классе **ConcurrentHashMap**) пустые значения **null** ключей или значений не допускаются. А в тех реализациях отображений, где они разрешаются (например, в классе **HashMap**), пользоваться пустыми значениями **null** следует очень аккуратно. Ведь целый ряд методов интерпретирует пустое значение **null** как признак отсутствия записи в отображении или потребность удалить ее.

СОВЕТ. Иногда ключи в отображении требуется представить не в порядке сортировки, а в другом порядке. Например, в каркасе JavaServer Faces с помощью отображения указываются метки и значения в окне выбора. Пользователи будут неприятно удивлены, если варианты выбора окажутся отсортированными по алфавиту (например, воскресенье, вторник, понедельник, пятница, среда, суббота, четверг) или хеш-коду. В таком случае следует воспользоваться классом **LinkedHashMap**, запоминающим порядок, в котором записи вводились в отображение, и перебирающим их в этом порядке.

7.5. Другие коллекции

В последующих разделах вкратце обсуждаются классы ряда других коллекций. Они могут оказаться полезными в практике программирования на Java.

7.5.1. Свойства

Класс `Properties` реализует отображение, которое легко сохраняется и загружается в простом текстовом формате. Такие отображения обычно применяются для хранения вариантов конфигурации или свойств прикладных программ, как демонстрируется в следующем примере кода:

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
try (OutputStream out = Files.newOutputStream(path)) {
  settings.store(out, "Program Properties");
}
```

В итоге получается следующий файл конфигурации программы:

```
#Свойства программы
#Mon Nov 03 20:52:33 CET 2014
width=200
title=Hello, World\!
```

ВНИМАНИЕ. Файлы свойств содержат текст в коде ASCII, а не в кодировке UTF-8. Комментарии начинаются со знака **#** или **!**. Символы с кодом меньше **'\u0021'** и больше **'\u007e'** обозначаются в Юникоде как экранируемые по следующему образцу: ***unnnn***. Знак новой строки в ключе или значении обозначается как **\n**. А знаки ****, **#**, **!** экранируются как ****, **\#**, **\!**.

Чтобы загрузить свойства из файла конфигурации, достаточно сделать следующий вызов:

```
try (InputStream in = Files.newInputStream(path)) {
  settings.load(in);
}
```

А для получения значения по заданном ключу следует вызвать метод `getProperty()`. Значение, используемое по умолчанию в отсутствие ключа, можно указать следующим образом:

```
String title = settings.getProperty("title", "New Document");
```

НА ЗАМЕТКУ. Исторически сложилось так, что класс **Properties** реализует интерфейс **Map<Object, Object>**, несмотря на то, что значения в отображении, которое он представляет, всегда являются символьными строками. Следовательно, для их получения *не* следует пользоваться методом **get()**, поскольку он возвращает значение в виде объекта типа **Object**.

Метод System.getProperties() возвращает объект типа Properties с системными свойствами. Наиболее употребительные системные свойства перечислены в табл. 7.7.

Таблица 7.7. Наиболее употребительные системные свойства

Ключ свойства	Описание
user.dir	Текущий рабочий каталог для данной виртуальной машины
user.home	Начальный каталог пользователя
user.name	Наименование учетной записи пользователя
java.version	Исполняемая версия Java для данной виртуальной машины
java.home	Начальный каталог установки Java
java.class.path	Путь к файлу класса, с помощью которого была запущена данная виртуальная машина
java.io.tmpdir	Каталог для хранения временных файлов (например, **/tmp**)
os.name	Наименование операционной системы (например, **Linux**)
os.arch	Архитектура операционной системы (например, **amd64**)
os.version	Версия операционной системы (например, **3.13.0-34-generic**)
file.separator	Разделитель файлов (знак / в Unix, знак \ в Windows)
path.separator	Разделитель путей к файлам (знак : в Unix, знак ; в Windows)
line.separator	Разделитель новых строк (знаки **\n** в Unix, знаки **\r\n** в Windows)

7.5.2. Множества битов

Класс BitSet представляет множество для хранения последовательности битов. Множество битов размещает их в массиве значений типа long, и поэтому пользоваться таким множеством эффективнее, чем массивом логических значений типа boolean. Множества битов удобны для хранения последовательностей двоичных разрядов признаков (или так называемых флагов) или для представления множеств неотрицательных целочисленных значений, где i-й бит, равный 1, обозначает, что значение i содержится в множестве.

В классе BitSet предоставляются удобные методы для получения и установки отдельных битов в множестве. Это намного проще, чем манипулировать отдельными битами для их хранения в переменных типа int или long. Имеются также методы, манипулирующие всеми битами в таких операциях с множествами, как объединение и пересечение. Полный перечень методов из класса BitSet приведен в табл. 7.8. Следует, однако, иметь в виду, что класс BitSet не относится к категории классов коллекций, поскольку он не реализует интерфейс Collection<Integer>.

Таблица 7.8. Методы из класса `BitSet`

Метод	Описание
`BitSet()`	Строят множество битов, которое может первоначально содержать `64` бита
`BitSet(int nbits)`	или заданное количество `nbits` битов
`void set(int bitIndex)`	Устанавливают отдельный бит в `1`, или указанное значение `value` по заданному индексу,
`void set(int fromIndex, int toIndex)`	или в пределах от `fromIndex` (включительно)
`void set(int bitIndex, boolean value)`	до `toIndex` (исключительно)
`void set(int fromIndex, int toIndex, boolean value)`	Сбрасывают в `0` отдельный бит по заданному индексу
`void clear(int bitIndex)`	в пределах от `fromIndex` (включительно)
`void clear(int fromIndex, int toIndex)`	до `toIndex` (исключительно)
`void clear()`	или же все биты
`void flip(int bitIndex)`	Сменяют на обратное состояние отдельного бита по заданному индексу или
`void flip(int fromIndex, int toIndex)`	в пределах от `fromIndex` (включительно) до `toIndex` (исключительно)
`boolean get(int bitIndex)`	Получают отдельный бит по заданному индексу или
`BitSet get(int fromIndex, int toIndex)`	в пределах от `fromIndex` (включительно) до `toIndex` (исключительно)
`int nextSetBit(int fromIndex)`	Возвращают индекс следующего или предыдущего бита,
`int previousSetBit(int fromIndex)`	устанавливаемого в `1` или сбрасываемого в `0`,
`int nextClearBit(int fromIndex)`	или же значение `-1`, если такой бит отсутствует
`int previousClearBit(int fromIndex)`	
`void and(Bitset set)`	Образует пересечение, разность,
`void andNot(Bitset set)`	объединение, симметричную разность
`void or(Bitset set)`	с заданным множеством `set`
`void xor(Bitset set)`	
`int cardinality()`	Возвращает количеств битов, установленных в данном множестве в `1`. *Предупреждение*: метод `size()` возвращает текущую длину битового вектора, а не размер множества
`byte[] toByteArray[]`	Размещают биты из данного множества
`long[] toByteArray[]`	в массиве
`IntStream stream()`	Возвращают поток данных или символьную строку целочисленных значений в данном множестве
`String toString()`	(т.е. индексы битов, установленных в `1`)

Окончание табл. 7.8

Метод	Описание
`static BitSet valueOf(byte[] bytes)`	Получают множество битов, содержащее предоставляемые биты
`static BitSet valueOf(long[] longs)`	
`static BitSet valueOf(ByteBuffer bb)`	
`static BitSet valueOf(LongBuffer lb)`	
`boolean isEmpty()`	Проверяют, является ли данное множество пустым
`boolean intersects(BitSet set)`	или содержит элемент, общий с заданным множеством *set*

7.5.3. Перечислимые множества и отображения

Для накопления множеств значений перечислимого типа вместо класса `BitSet` используется класс `EnumSet`. У класса `EnumSet` отсутствуют открытые конструкторы. Для построения перечислимого множества служит статический фабричный метод, как показано ниже. Для обращения с множеством типа `EnumSet` можно также пользоваться методами из интерфейса `Set`.

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
               FRIDAY, SATURDAY, SUNDAY };
Set<Weekday> always = EnumSet.allOf(Weekday.class);
Set<Weekday> never = EnumSet.noneOf(Weekday.class);
Set<Weekday> workday = EnumSet.range(Weekday.MONDAY,
                                     Weekday.FRIDAY);
Set<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY,
                              Weekday.FRIDAY);
```

Класс `EnumMap` представляет отображение с ключами, относящимися к перечислимому типу. Такое отображение реализуется как массив значений. Тип ключей указывается в конструкторе данного класса следующим образом:

```
EnumMap<Weekday, String> personInCharge = new EnumMap<>(Weekday.class);
personInCharge.put(Weekday.MONDAY, "Fred");
```

7.5.4. Стеки и разнотипные очереди

Стек — это структура данных для ввода и удаления элементов с одного конца, называемого "вершиной" стека. Обычная очередь позволяет вводить в нее элементы с одного конца, называемого "хвостом", и удалять их с другого конца, называемого "головой" очереди. Очередь с двухсторонним доступом (или двухсторонняя очередь) допускает ввод и удаление элементов с обоих ее концов. Ввод элементов посредине всех этих структур данных не предусмотрен.

Методы для обращения с упомянутыми выше структурами данных определяются в интерфейсах `Queue` и `Deque`. Но для стека в каркасе коллекций предусмотрен не интерфейс, а только класс `Stack`, внедренный на самой ранней стадии существования Java, и поэтому пользоваться им не рекомендуется. Если требуется организовать стек,

обычную или двухстороннюю очередь, а потоковая безопасность особого значения не имеет, то для этой цели можно воспользоваться классом ArrayDeque.

Для обращения со стеком служат методы push() и pop(), как показано ниже.

```
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("Peter");
stack.push("Paul");
stack.push("Mary");
while (!stack.isEmpty())
  System.out.println(stack.pop());
```

А для обращения с обычной очередью служат методы add() и remove(), как демонстрируется в следующем примере кода:

```
Queue<String> queue = new ArrayDeque<>();
queue.add("Peter");
queue.add("Paul");
queue.add("Mary");
while (!queue.isEmpty())
  System.out.println(queue.remove());
```

Потокобезопасные очереди обычно употребляются в параллельно выполняемых программах. Подробнее об этом речь пойдет в главе 10.

Очередь с приоритетами допускает извлечение ее элементов в отсортированном порядке после их ввода в произвольном порядке. Это означает, что всякий раз, когда вызывается метод remove(), получается наименьший элемент, находящийся в настоящий момент в очереди с приоритетами.

Очередь с приоритетами чаще всего применяется для планирования заданий, как показано в приведенном ниже примере кода. У каждого задания имеется свой приоритет. Задания вводятся в такую очередь в произвольном порядке. Всякий раз, когда новое задание запускается на выполнение, из очереди удаляется задание с наивысшим приоритетом. (По традиции наивысший приоритет обозначается числом 1, и поэтому в результате вызова метода remove() из очереди извлекается наименьший элемент.)

```
public class Job implements Comparable<Job> { ... }
...
PriorityQueue<Job> jobs = new PriorityQueue<>();
jobs.add(new Job(4, "Collect garbage"));
jobs.add(new Job(9, "Match braces"));
jobs.add(new Job(1, "Fix memory leak"));
...
while (jobs.size() > 0) {
  Job job = jobs.remove(); // Наиболее срочные задания извлекаются
                           // из очереди первыми
  execute(job);
}
```

Аналогично древовидному множеству типа TreeSet, очередь с приоритетами может содержать элементы класса, реализующего интерфейс Comparable. С другой стороны, конструктору такого класса может быть передан интерфейс Comparator. Но в отличие от древовидного множества типа TreeSet, перебор элементов очереди с

приоритетами совсем не обязательно приводит к их получению в отсортированном порядке. В очереди с приоритетами применяются алгоритмы для ввода и удаления из нее элементов, в результате чего наименьший элемент тяготеет к корню очереди и экономится время на сортировку всех элементов.

7.5.5. Слабые хеш-отображения

Класс `WeakHashMap` предназначен для решения следующей интересной задачи: что произойдет со значением, ключ которого не употребляется нигде больше в программе? Если последняя ссылка на ключ исчезнет, то обратиться к объекту-значению не удастся, и поэтому такой объект должен быть удален сборщиком "мусора".

Но сделать это не так-то просто. Ведь сборщик "мусора" отслеживает действующие объекты. А поскольку объект отображения действует, то действуют и все записи в нем. Они не утилизируются, как, впрочем, и значения, на которые они ссылаются.

Именно эту задачу и решает класс `WeakHashMap`. Его структура данных взаимодействует со сборщиком "мусора" для удаления пар "ключ–значение", когда единственная ссылка на ключ присутствует в записи из хеш-таблицы.

Формально для хранения ключей в классе `WeakHashMap` применяются так называемые *слабые ссылки*. В объекте типа `WeakReference` хранится ссылка на другой объект (в данном случае — ключ к хеш-таблице). Объекты данного типа интерпретируются сборщиком "мусора" как особый случай. Если объект доступен *только* по слабой ссылке, сборщик "мусора" утилизирует его, ставя слабую ссылку в очередь, связанную с объектом типа `WeakReference`. Всякий раз, когда для этого объекта вызывается метод, в слабом хеш-отображении типа `WeakHashMap` проверяется наличие новых поступлений в его очереди слабых ссылок и затем из него удаляются связанные с ними записи.

7.6. Представления

Представление коллекции является легковесным объектом, реализующим интерфейс коллекции, но не хранящим его элементы. Например, методы `keySet()` и `values()` из отображения дают его представления.

Другим примером служит метод `Arrays.asList()`. Так, если массив `a` относится к типу `E[]`, то в результате вызова `Arrays.asList(a)` возвращается обобщенный список типа `List<T>`, поддерживаемый элементами данного массива.

Как правило, в представлении не поддерживаются все операции его интерфейса. Нет, например, никакого смысла вызывать метод `add()` для набора ключей из отображения или списка, возвращаемого методом `Arrays.asList()`. В последующих разделах рассматриваются некоторые разновидности представлений, предоставляемых каркасом коллекций в Java.

7.6.1. Диапазоны

Из списка можно составить представление его подсписка, как показано в следующем примере кода:

```
List<String> sentence = ...;
List<String> nextFive = sentence.subList(5, 10);
```

В этом представлении элементы доступны по индексу в пределах от 5 до 9. Любые изменения в подсписке, в том числе установка, ввод или удаление элементов, оказывают влияние на исходный список.

Для сортируемых множеств и отображений диапазон указывается по нижнему и верхнему пределам, как показано ниже. Как и в методе `subList()`, первый предел указывается включительно, а второй — исключительно.

```
TreeSet<String> words = ...;
SortedSet<String> asOnly = words.subSet("a", "b");
```

Методы `headSet()` и `tailSet()` возвращают поддиапазон без нижнего и верхнего пределов, как, например, в следующей строке кода:

```
NavigableSet<String> nAndBeyond = words.tailSet("n");
```

В методах из интерфейса `NavigableSet` можно выбрать задание каждого предела включительно или исключительно (см. табл. 7.5). А для сортируемого отображения имеются аналогичные методы `subMap()`, `headMap()` и `tailMap()`.

7.6.2. Пустые и одноэлементные представления

В классе `Collections` имеются статические методы для получения неизменяемого пустого списка, множества, сортируемого множества, навигационного множества, итератора, списочного итератора или перечисления (интерфейса, похожего на итератор и унаследованного из версии Java 1.0).

Аналогично имеются статические методы для получения множества или списка с единственным элементом, а также отображения с единственной парой "ключ–значение" (см. табл. 7.3).

Так, если в методе требуется отображение атрибутов и такие атрибуты отсутствуют или имеется только один из них, то можно сделать следующий вызов:

```
doWork(Collections.emptyMap());
```

или такой вызов:

```
doWork(Collections.singletonMap("id", id));
```

вместо того, чтобы создавать тяжеловесный объект типа `HashMap` или `TreeMap`.

7.6.3. Неизменяемые представления

Иногда содержимое коллекции требуется сделать общедоступным, но не изменять его. Разумеется, значения из исходной коллекции можно скопировать в новую коллекцию, но это потенциально затратная операция. Вместо этого лучше выбрать неизменяемое представление. В качестве примера рассмотрим типичную ситуацию, где в объекте типа `Person` хранится список друзей конкретного лица. Если бы метод `getFriends()` возвратил ссылку на этот список, то его можно было бы изменить в

вызывающем коде. Но для этой цели надежнее предоставить неизменяемое представление следующим образом:

```
public class Person {
  private ArrayList<Person> friends;

  public List<Person> getFriends() {
    return Collections.unmodifiableList(friends);
  }
  ...
}
```

Все модифицирующие методы генерируют исключение, если они вызываются для неизменяемого представления. Как следует из табл. 7.3, неизменяемые представления можно получить в виде коллекций, списков, множеств, сортируемых множеств, навигационных множеств, отображений, сортируемых отображений и навигационных отображений.

НА ЗАМЕТКУ. Как было показано в главе 6, элементы неверного типа можно незаконным путем переместить в обобщенную коллекцию (это явление называется *загрязнением "кучи"*), а ошибка времени выполнения возникнет при попытке извлечь, а не ввести элемент неподходящего типа в такую коллекцию. Чтобы устранить подобную ошибку, следует воспользоваться *проверяемым представлением*. Например, в том месте, где строится списочный массив типа **`ArrayList<String>`**, следует построить проверяемое представление следующим образом:

```
List<String> strings =
     Collections.checkedList(new ArrayList<>(), String.class);
```

Такое представление контролирует все операции ввода элементов в список и генерирует исключение, если в него вводится объект неверного типа.

НА ЗАМЕТКУ. Средствами класса **`Collections`** можно также получать *синхронизированные* представления, обеспечивающие надежный параллельный доступ к структурам данных. На практике такие представления не настолько полезны, как структуры данных из пакета **`java.util.concurrent`**, специально предназначенные для параллельного доступа. Поэтому лучше пользоваться классами из данного пакета, держась подальше от синхронизированных представлений.

Упражнения

1. Реализуйте алгоритм под названием "Решето Эратосфена" для выявления всех простых чисел меньше или равных ***n***. Сначала введите все числа от 2 до ***n*** в множество. Затем организуйте повторяющийся поиск наименьшего элемента ***s*** в этом множестве, постепенно удаляя из него элементы s^2, $s \cdot (s + 1)$, $s \cdot (s + 2)$ и т.д. Этот поиск завершается при условии, когда $s^2 > n$. Реализуйте данный алгоритм как для множества типа `HashSet<Integer>`, так и для множества типа `BitSet`.
2. Сделайте все буквы прописными в символьных строках, содержащихся в массиве. С этой целью воспользуйтесь сначала итератором, затем перебором индексных значений в цикле и, наконец, методом `replaceAll()`.

3. Как вычислить объединение, пересечение и разность двух множеств, используя только методы из интерфейса `Set`, но не организуя циклы?
4. Воспроизведите ситуацию, когда возникает исключение типа `ConcurrentModificationException`. Что можно предпринять, чтобы избежать этого?
5. Реализуйте метод `public static void swap(List<?> list, int i, int j)`, выполняющий перестановку элементов обычным образом, когда класс, определяющий тип параметра `list`, реализует интерфейс `RandomAccess`, а иначе сводящий к минимуму обход элементов на позициях, обозначаемых индексами `i` и `j`.
6. В этой главе рекомендовалось пользоваться интерфейсами вместо конкретных классов структур данных, например, интерфейсом `Map` вместо класса `TreeMap`. К сожалению, подобная рекомендация не идет дальше этого. Почему нельзя, например, воспользоваться интерфейсом `Map<String, Set<Integer>>`, чтобы представить содержание документа? (*Подсказка:* как инициализировать его?) Каким типом структуры данных можно вместо этого воспользоваться?
7. Напишите программу для чтения всех слов из файла и вывода частоты, с которой каждое слово встречается в нем. Воспользуйтесь для этой цели классом `TreeMap<String, Integer>`.
8. Напишите программу для чтения всех слов из файла и вывода строк, в которых каждое слово встречается в нем. Воспользуйтесь для этой цели преобразованием из символьных срок в множества.
9. Счетчик в отображении счетчиков можно обновить следующим образом:

   ```
   counts.merge(word, 1, Integer::sum);
   ```

   Сделайте то же самое без метода `merge()`, воспользовавшись, во-первых, методом `contains()`; во-вторых, методом `get()` и проверкой пустых значений (`null`); в-третьих, методом `getOrDefault()` и, в-четвертых, методом `putIfAbsent()`.
10. Реализуйте алгоритм Дейкстры для поиска кратчайших путей между городами, связанными сетью автомобильных дорог. (Описание этого алгоритма можно найти в популярной литературе по алгоритмам или в соответствующей статье Википедии.) Воспользуйтесь вспомогательным классом `Neighbor` для хранения названия соседнего города и расстояния до него. Представьте полученный граф в виде преобразования названий городов в множества соседних городов. Воспользуйтесь в данном алгоритме классом `PriorityQueue<Neighbor>`.
11. Напишите программу для чтения предложения в списочный массив. Затем перетасуйте в массиве все слова, кроме первого и последнего, используя метод `Collections.shuffle()`, но не копируя слова в другую коллекцию.
12. Используя метод `Collections.shuffle()`, напишите программу для чтения предложения, перетасовки его слов и вывода результата. Устраните (до и после перетасовки) написание начального слова с заглавной буквы и наличие точки в конце предложения. *Подсказка:* не перетасовывайте при этом слова.

13. Всякий раз, когда в отображение типа `LinkedHashMap` вводится новый элемент, вызывается метод `removeEldestEntry()`. Реализуйте подкласс `Cache`, производный от класса `LinkedHashMap` и ограничивающий размер отображения заданной величиной, предоставляемой в конструкторе этого класса.
14. Напишите метод для получения неизменяемого представления списка чисел от 0 до n, не сохраняя эти числа.
15. Обобщите предыдущее упражнение произвольным функциональным интерфейсом `IntFunction`. Имейте в виду, что в конечном итоге может получиться бесконечная коллекция, поэтому некоторые методы (например, `size()` и `toArray()`) должны генерировать исключение типа `UnsupportedOperationException`.
16. Усовершенствуйте реализацию из предыдущего упражнения, организовав кеширование последних 100 значений, вычисленных функцией.
17. Покажите, каким образом проверяемое представление может сообщить о конкретной ошибке, ставшей причиной загрязнения "кучи".
18. В классе `Collections` имеются статические константы `EMPTY_LIST`, `EMPTY_MAP` и `EMPTY_SET`. Почему они не так полезны, как методы `emptyList()`, `emptyMap()` и `emptySet()`?

Глава 8

Потоки данных

В этой главе...

Потоки данных обеспечивают представление данных, позволяющее указать вычисления на более высоком концептуальном уровне, чем коллекции. С помощью потока данных можно указать, что и как именно требуется сделать с данными, а планирование операций предоставить конкретной реализации. Допустим, что требуется вычислить среднее некоторого свойства. С этой целью указывается источник данных и свойство, а средствами библиотеки потоков данных можно оптимизировать вычисление, используя, например, несколько потоков исполнения для расчета сумм, подсчета и объединения результатов.

Основные положения этой главы приведены ниже.

1. Итераторы подразумевают конкретную методику обхода и препятствуют организации эффективного параллельного выполнения.
2. Потоки данных можно создавать из коллекций, массивов, генераторов или итераторов.
3. Для отбора элементов из потока данных служит метод `filter()`, а для их преобразования — метод `map()`.
4. Другие операции преобразования потоков данных реализуются методами `limit()`, `distinct()` и `sorted()`.
5. Для получения результата из потока данных служат операции сведения, реализуемые, в частности, методами `count()`, `max()`, `min()`, `findFirst()` и `findAny()`. Некоторые из этих методов возвращают необязательное значение типа `Optional`.
6. Тип `Optional` специально предназначен в качестве надежной альтернативы обработке пустых значений `null`. Для надежного применения этого типа служат методы `ifPresent()` и `orElse()`.
7. Результаты, получаемые из потоков данных, можно накапливать в коллекциях, массивах, символьных строках или отображениях.
8. Методы `groupingBy()` и `partitioningBy()` из класса `Collectors` позволяют разделять содержимое потоков данных на группы и получать результат для каждой группы в отдельности.
9. Для примитивных типов `int`, `long` и `double` имеются специализированные потоки данных.
10. Параллельные потоки данных автоматически распараллеливают потоковые операции.

8.1. От итерации к потоковым операциям

Для обработки коллекции обычно требуется перебрать ее элементы и выполнить над ними некоторую операцию. Допустим, что требуется подсчитать все длинные слова в книге. Сначала организуем их вывод списком следующим образом:

```
String contents = new String(Files.readAllBytes(
  Paths.get("alice.txt")), StandardCharsets.UTF_8);
```

```
    // прочитать текст из файла в символьную строку
List<String> words = Arrays.asList(contents.split("\\PL+"));
    // разбить полученную символьную строку на слова;
    // небуквенные символы считаются разделителями
```

А теперь можно перебрать слова таким образом:

```
int count = 0;
for (String w : words) {
  if (w.length() > 12) count++;
}
```

Ниже показано, каким образом аналогичная операция осуществляется с помощью потоков данных.

```
long count = words.stream()
    .filter(w -> w.length() > 12)
    .count();
```

В последнем случае не нужно искать в цикле наглядного подтверждения операций фильтрации и подсчета слов. Сами имена методов свидетельствуют о том, что именно предполагается сделать в коде. Более того, если в цикле во всех подробностях предписывается порядок выполнения операций, то в потоке данных операции можно планировать как угодно, при условии, что будет достигнут правильный результат.

Достаточно заменить метод `stream()` на метод `parallelStream()`, чтобы организовать средствами библиотеки потоков данных параллельное выполнение операций фильтрации и подсчета слов, как показано ниже.

```
long count = words.parallelStream()
    .filter(w -> w.length() > 12)
    .count();
```

Потоки данных действуют по принципу "что, а не как делать". В рассматриваемом здесь примере мы описываем, что нужно сделать: получить длинные слова и подсчитать их. При этом мы не указываем, в каком порядке или потоке исполнения это должно произойти. Напротив, в упомянутом выше цикле точно указывается порядок организации вычислений, а следовательно, исключается всякая возможность для оптимизации.

На первый взгляд поток данных похож на коллекцию, поскольку он позволяет преобразовывать и извлекать данные. Но у потока данных имеются следующие существенные отличия:

1. Поток данных не сохраняет свои элементы. Они могут храниться в основной коллекции или формироваться по требованию.
2. Потоковые операции не изменяют их источник. Например, метод `filter()` не удаляет элементы из нового потока данных, но выдает новый поток, в котором они отсутствуют.
3. Потоковые операции выполняются *по требованию*, когда это возможно. Это означает, что они не выполняются до тех пор, пока не потребуется их результат. Так, если требуется подсчитать только пять длинных слов вместо всех слов,

метод filter() прекратит фильтрацию после пятого совпадения. Следовательно, потоки данных могут быть бесконечными!

Вернемся к предыдущему примеру, чтобы рассмотреть его подробнее. Методы stream() и parallelStream() выдают *поток данных* для списка слов words. А метод filter() возвращает другой поток данных, содержащий только те слова, длина которых больше 12 букв. И наконец, метод count() сводит этот поток данных в конечный результат.

Такая последовательность операций весьма характерна для обращения с потоками данных. Конвейер операций организуется в следующие три стадии.

1. Создание потока данных.
2. Указание *промежуточных операций* для преобразования исходного потока данных в другие потоки, возможно, в несколько этапов.
3. Выполнение *оконечной операции* для получения результата. Эта операция принуждает к выполнению по требованию тех операций, которые ей предшествуют. А впоследствии поток данных может больше не понадобиться.

В рассматриваемом здесь примере поток данных был создан методом stream() или parallelStream(). Метод filter() преобразовал его, а метод count() выполнил оконечную операцию.

В следующем разделе будет показано, как создается поток данных. В трех последующих разделах рассматриваются потоковые преобразования, а в пяти следующих за ними разделах — оконечные операции.

8.2. Создание потока данных

Как было показано выше, любую коллекцию можно преобразовать в поток данных методом stream() из интерфейса Collection. Если же вместо коллекции имеется массив, то для этой цели служит метод Stream.of(), как показано ниже.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
    // Метод split() возвращает массив типа String[]
```

У метода of() имеются аргументы переменной длины, и поэтому поток данных можно построить из любого количества аргументов, как показано ниже. А для создания потока данных из части массива служит метод Arrays.stream(array, from, to).

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Чтобы создать поток данных без элементов, достаточно вызвать статический метод Stream.empty() следующим образом:

```
Stream<String> silence = Stream.empty();
    // Обобщенный тип <String> выводится автоматически;
    // что равнозначно вызову Stream.<String>empty()
```

Для создания бесконечных потоков данных в интерфейсе Stream имеются два статических метода. В частности, метод generate() принимает функцию без аргументов

(а формально — объект функционального интерфейса `Supplier<T>`; см. раздел 3.6.2). Всякий раз, когда требуется потоковое значение, эта функция вызывается для получения данного значения. Например, поток постоянных значений можно получить следующим образом:

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

а поток случайных чисел таким образом:

```
Stream<Double> randoms = Stream.generate(Math::random);
```

А для получения бесконечных последовательностей вроде **0 1 2 3 ...** служит метод `iterate()`. Этот метод принимает начальное значение и функцию (а формально — объект функционального интерфейса `UnaryOperator<T>`) и повторно применяет функцию к предыдущему результату, как показано в следующем примере кода:

```
Stream<BigInteger> integers =
      Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

Первым элементом такой последовательности является начальное значение `BigInteger.ZERO`; вторым ее элементом — значение, получаемое в результате вызова функции `f(seed)`, или **1** (как крупное целочисленное значение); следующим элементом — значение, получаемое в результате вызова функции `f(f(seed))`, или **2** и т.д., где `seed` — начальное значение.

НА ЗАМЕТКУ. В прикладном программном интерфейсе Java API имеется целый ряд методов, возвращающих потоки данных. Так, в классе **`Pattern`** имеется метод **`splitAsStream()`**, разделяющий последовательность символов типа **`CharSequence`** по регулярному выражению. Например, для разделения символьной строки на отдельные слова можно воспользоваться следующим оператором:

```
Stream<String> words = Pattern.compile("\\PL+").splitAsStream(contents);
```

А статический метод **`Files.lines()`** возвращает поток данных типа **`Stream`**, содержащий все строки из файла, как показано ниже.

```
try (Stream<String> lines = Files.lines(path)) {
 Обработать строки
}
```

8.3. Методы `filter()`, `map()` и `flatMap()`

В результате преобразования потока данных получается другой поток данных, элементы которого являются производными от элементов исходного потока. Ранее демонстрировалось преобразование методом `filter()`, в результате которого получается новый поток данных с элементами, удовлетворяющими определенному условию. А в приведенном ниже примере кода поток символьных строк преобразуется в другой поток, содержащий только длинные слова. В качестве аргумента метода `filter()` указывается объект типа `Predicate<T>`, т.е. функция, преобразующая тип **`T`** в логический тип `boolean`.

```
List<String> words = ...;
Stream<String> longWords = words.stream().filter(w -> w.length() > 12);
```

Нередко значения в потоке данных требуется каким-то образом преобразовать. Для этой цели можно воспользоваться методом `map()`, передав ему функцию, которая и выполняет нужное преобразование. Например, буквы во всех словах можно сделать строчными следующим образом:

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

В данном примере методу `map()` была передана ссылка на метод. Но вместо нее нередко передается *лямбда-выражение*, как показано ниже. Получающийся в итоге поток данных содержит первую букву каждого слова.

```
Stream<String> firstLetters = words.stream().map(s -> s.substring(0, 1));
```

При вызове метода `map()` передаваемая ему функция применяется к каждому элементу потока данных, в результате чего образуется новый поток данных с полученными результатами. А теперь допустим, что имеется метод, возвращающий не одно значение, а поток значений, как показано ниже. Например, в результате вызова `letters("boat")` образуется поток данных `["b", "o", "a", "t"]`.

```
public static Stream<String> letters(String s) {
  List<String> result = new ArrayList<>();
  for (int i = 0; i < s.length(); i++)
    result.add(s.substring(i, i + 1));
  return result.stream();
}
```

НА ЗАМЕТКУ. Приведенный выше метод **`letters()`** можно реализовать намного изящнее с помощью метода **`IntStream.range()`**, рассматриваемого далее, в разделе 8.13. См. также упражнение 5 в конце этой главы.

Допустим, что метод `letters()` передается методу `map()` для преобразования потока символьных строк следующим образом:

```
Stream<Stream<String>> result = words.stream().map(w -> letters(w));
```

В итоге получится поток потоков символьных строк вроде `[... ["y", "o", "u", "r"], ["b", "o", "a", "t"], ...]`. Чтобы свести его к потоку букв `[... "y", "o", "u", "r", "b", "o", "a", "t", ...]`, вместо метода `map()` следует вызвать метод `flatMap()` таким образом:

```
Stream<String> flatResult = words.stream().flatMap(w -> letters(w))
    // Вызывает метод letters() для каждого слова и сводит результаты
```

НА ЗАМЕТКУ. Аналогичный метод **`flatMap()`** можно обнаружить и в других классах, а не в тех, что представляют потоки данных. Это общий принцип вычислительной техники. Допустим, что имеется обобщенный тип `G` (например, **`Stream`**) и функции **`f()`** и **`g()`**, преобразующие некоторый тип `T` в тип `G<U>`, а тип `U` — в тип `G<V>` соответственно. В таком случае эти функции можно составить

вместе, используя метод **flatMap()**, т.е. применить сначала функцию **f()**, а затем функцию **g()**. В этом состоит главная идея теории *монад*. Впрочем, метод **flatMap()** можно применять, и не зная ничего о монадах.

8.4. Извлечение и соединение потоков данных

В результате вызова *поток*.limit(n) возвращается поток данных, оканчивающийся после n элементов или по завершении исходного потока данных, если тот короче. Метод limit() особенно удобен для ограничения бесконечных потоков данных до определенной длины. Так, в следующей строке кода получается поток данных, состоящий из 100 произвольных чисел:

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

А в результате вызова *поток*.skip(n) происходит совершенно противоположное: отбрасываются первые n элементов. Если вернуться к рассмотренному ранее примеру чтения текста книги, то в силу особенностей работы метода split() первым элементом потока данных оказывается нежелательная пустая строка. От нее можно избавиться, вызвав метод skip() следующим образом:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

Два потока данных можно соединить вместе с помощью статического метода concat() из интерфейса Stream, как показано ниже. Разумеется, первый из этих потоков не должен быть бесконечным, иначе второй поток вообще не сможет соединиться с ним.

```
Stream<String> combined = Stream.concat(
letters("Hello"), letters("World"));
  // Получается следующий поток данных:
  // ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

8.5. Другие потоковые преобразования

Метод distinct() возвращает поток данных, получающий свои элементы из исходного потока данных в том же самом порядке, за исключением того, что дубликаты в нем подавляются. Дубликаты не должны быть смежными, как показано ниже.

```
Stream<String> uniqueWords =
  Stream.of("merrily", "merrily", "merrily", "gently").distinct();
  // В итоге возвращается только одна строка "merrily"
```

Для сортировки потоков данных имеется несколько вариантов метода sorted(). Один из них служит для обработки потоков данных, состоящих из элементов типа Comparable, а другой принимает в качестве параметра компаратор типа Comparator. Так, в следующем примере кода символьные строки сортируются таким образом, чтобы первой в потоке данных следовала самая длинная строка:

```
Stream<String> longestFirst =
  words.stream().sorted(Comparator.comparing(String::length).reversed());
```

Как и во всех остальных потоковых преобразованиях, метод `sorted()` выдает новый поток данных, элементы которого происходят из исходного потока и располагаются в отсортированном порядке. Разумеется, коллекцию можно отсортировать, не прибегая к потокам данных. Метод `sorted()` удобно применять в том случае, если процесс сортировки является частью поточного конвейера.

И наконец, метод `peek()` выдает другой поток данных с теми же самыми элементами, что и у исходного потока, но передаваемая ему функция вызывается всякий раз, когда извлекается элемент. Это удобно для целей отладки, как показано ниже.

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20).toArray();
```

Сообщение выводится в тот момент, когда элемент доступен в потоке данных. Подобным образом можно проверить, что бесконечный поток данных, возвращаемый методом `iterate()`, обрабатывается по требованию. Для целей отладки в методе `peek()` можно вызвать метод, в котором устанавливается точка прерывания.

8.6. Простые методы сведения

А теперь, когда было показано, каким образом осуществляется создание и преобразование потоков данных, мы наконец-то добрались до самого главного — получения ответов на запросы данных из потоков. В этом разделе рассматриваются так называемые *методы сведения*. Они выполняют *оконечные операции*, сводя поток данных к непотоковому значению, которое может быть далее использовано в программе. Ранее уже демонстрировался простой метод сведения `count()`, возвращающий количество элементов в потоке данных.

К числу других простых методов сведения относятся методы `max()` и `min()`, возвращающие наибольшее и наименьшее значения соответственно. Но не все так просто, поскольку эти методы на самом деле возвращают значение типа `Optional<T>`, которое заключает в себе ответ на запрос данных из потока или обозначает, что запрашиваемые данные отсутствуют, поскольку поток оказался пустым. Раньше в подобных случаях возвращалось пустое значение `null`. Но это могло привести к исключениям в связи с пустыми указателями в не полностью протестированной программе. Тип `Optional` удобнее для обозначения отсутствующего возвращаемого значения. Более подробно тип `Optional` рассматривается в следующем разделе. Ниже показано, как получить максимальное значение из потока данных.

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.getOrElse(""));
```

Метод `findFirst()` возвращает первое значение из непустой коллекции. Зачастую он применяется вместе с методом `filter()`. Так, в следующем примере кода обнаруживается первое слово, начинающееся с буквы `Q`:

```
Optional<String> startsWithQ =
    words.filter(s -> s.startsWith("Q")).findFirst();
```

Если же требуется *любое* совпадение, а не только первое, то следует воспользоваться методом `findAny()`, как показано ниже. Это оказывается эффективным при распараллеливании потока данных, поскольку поток может известить о любом обнаруженном в нем совпадении, вместо того чтобы ограничиваться только первым совпадением.

```
Optional<String> startsWithQ =
    words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

Если же требуется только выяснить, имеется *ли* вообще совпадение, то следует воспользоваться методом `anyMatch()`, как показано ниже. Этот метод принимает предикатный аргумент, и поэтому ему не требуется метод `filter()`.

```
boolean aWordStartsWithQ =
   words.parallel().anyMatch(s -> s.startsWith("Q"));
```

Имеются также методы `allMatch()` и `noneMatch()`, возвращающие логическое значение `true`, если с предикатом совпадают все элементы в потоке данных или не совпадает ни один из его элементов соответственно. Эти методы также выгодно выполнять в параллельном режиме.

8.7. Тип `Optional`

Объект типа `Optional<T>` служит оболочкой для объекта обобщенного типа `T` или же ни для одного из объектов. В первом случае считается, что значение *присутствует*. Тип `Optional<T>` служит в качестве более надежной альтернативы ссылке на обобщенный тип `T`, которая делается на объект или оказывается пустой. Но этот тип надежнее, если правильно им пользоваться. В следующем разделе поясняется, как это делается.

8.7.1. Как обращаться с необязательными значениями

Для эффективного применения типа `Optional` самое главное — выбрать метод, который возвращает *альтернативный вариант*, если значение отсутствует, или *употребляет значение*, если только оно присутствует. Рассмотрим первую методику обращения с необязательными значениями. Нередко имеется значение, возможно, пустая строка `""`, которое требуется использовать по умолчанию в отсутствие совпадения:

```
String result = optionalString.orElse("");
  // Заключенная в оболочку строка, а в ее отсутствие - пустая строка ""
```

Кроме того, можно вызвать функцию для вычисления значения по умолчанию следующим образом:

```
String result = optionalString.orElseGet(() ->
                System.getProperty("user.dir"));
    // Функция вызывается только по мере надобности
```

С другой стороны, в отсутствие значения можно сгенерировать исключение таким образом:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
  // предоставить метод, который выдает объект исключения
```

В приведенных выше примерах было показано, как получить альтернативный вариант, если значение отсутствует. Другая методика обращения с необязательными значениями состоит в том, чтобы употребить значение, если только оно присутствует.

Метод `ifPresent()` принимает функцию в качестве аргумента, как показано ниже. Если необязательное значение существует, оно передается данной функции. В противном случае ничего не происходит.

```
optionalValue.ifPresent(v -> Обработать v);
```

Так, если в множество требуется ввести значение, при условии, что оно существует, достаточно сделать следующий вызов:

```
optionalValue.ifPresent(v -> results.add(v));
```

или просто

```
optionalValue.ifPresent(results::add);
```

В результате вызова метода `ifPresent()` передаваемая ему функция никакого значения не возвращает. Если же требуется обработать результат выполнения функции, следует вызвать метод `map()`, как показано ниже. В итоге переменная `added` будет содержать одно из следующих трех значений: логическое значение `true` или `false`, заключенное в оболочку типа `Optional`, если необязательно значение `optionalValue` присутствовало, а иначе — пустое значение типа `Optional`.

```
Optional<Boolean> added = optionalValue.map(results::add);
```

НА ЗАМЕТКУ. Метод **`map()`** служит аналогом метода **`map()`** из интерфейса **`Stream`**, упоминавшегося в разделе 8.3. Необязательное значение можно рассматривать в качестве потока данных нулевой или единичной длины. Получаемый в итоге результат также имеет нулевую или единичную длину. И в последнем случае применялась функция.

8.7.2. Как не следует обращаться с необязательными значениями

Если необязательные значения типа `Optional` не применяются правильно, то они не дают никаких преимуществ по сравнению с прежним подходом, предоставлявшим выбор между чем-то существующим или несуществующим, т.е. пустым (`null`). Метод `get()` получает заключенный в оболочку элемент значения типа `Optional`, если это значение существует, а иначе — генерирует исключение типа `NoSuchElementException`. Таким образом, следующий фрагмент кода:

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod()
```

не надежнее, чем такой код:

```
T value = ...;
value.someMethod();
```

Метод `isPresent()` извещает, содержит ли значение объект типа `Optional<T>`. Но выражение

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

не проще, чем выражение

```
if (value != null) value.someMethod();
```

8.7.3. Формирование необязательных значений

До сих пор обсуждалось, как употреблять объект типа `Optional`, созданный кем-то другим. Если же требуется написать метод, создающий объект типа `Optional`, то для этой цели имеется несколько статических методов. В приведенном ниже примере демонстрируется применение двух таких методов: `Optional.of(result)` и `Optional.empty()`.

```
public static Optional<Double> inverse(Double x) {
  return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

Метод `ofNullable()` служит в качестве моста между возможными пустыми (`null`) и необязательными значениями. Так, при вызове метода `Optional.ofNullable(obj)` возвращается результат вызова метода `Optional.of(obj)`, если объект `obj` не пустой, а иначе — результат вызова метода `Optional.empty()`.

8.7.4. Сочетание функций необязательных значений с методом `flatMap()`

Допустим, что имеется метод `f()`, возвращающий объект типа `Optional<T>`, а у целевого типа `T` — метод `g()`, возвращающий объект типа `Optional<U>`. Если бы это были обычные методы, их можно было бы составить в вызов `s.f().g()`. Но такое сочетание не годится, поскольку результат вызова `s.f()` относится к типу `Optional<T>`, а не к типу `T`. Вместо этого нужно сделать следующий вызов:

```
Optional<U> result = s.f().flatMap(T::g);
```

Если объект, получаемый в результате вызова `s.f()`, присутствует, то к нему применяется метод `g()`. В противном случае возвращается пустой объект типа `Optional<U>`.

Очевидно, что данный процесс можно повторить, если имеются другие методы или лямбда-выражения, возвращающие необязательные значения типа `Optional`. В таком случае из них можно составить конвейер, связав их вызовы в цепочку с методом `flatMap()`, который будет успешно завершен, если завершатся все остальные части конвейера.

В качестве примера рассмотрим надежный метод inverse() из предыдущего раздела. Допустим, что имеется также следующий надежный метод для извлечения квадратного корня:

```
public static Optional<Double> squareRoot(Double x) {
  return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

В таком случае извлечь квадратный корень из значения, возвращаемого методом inverse(), можно следующим образом:

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

или таким способом, если он предпочтительнее:

```
Optional<Double> result =
    Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::squareRoot);
```

Если метод inverse() или squareRoot() возвращает результат вызова метода Optional.empty(), то конечный результат окажется пустым.

НА ЗАМЕТКУ. Как было показано в разделе 8.3, метод **flatMap()** из интерфейса **Stream** служит для составления двух других методов, получающих потоки данных, сводя их в результирующий поток потоков. Аналогичным образом действует и метод **Optional.flatMap()**, если необязательное значение интерпретируется как поток данных нулевой или единичной длины.

8.8. Накопление результатов

По завершении обработки потока данных нередко требуется просмотреть полученные результаты. С этой целью можно вызвать метод iterate(), предоставляющий устаревший итератор, которым можно воспользоваться для обхода элементов. С другой стороны, можно вызвать метод forEach(), чтобы применить функцию к каждому элементу следующим образом:

```
stream.forEach(System.out::println);
```

В параллельном потоке данных метод forEach() выполняет обход элементов в произвольном порядке. Если же их требуется обработать в потоковом порядке, то следует вызвать метод forEachOrdered(). Разумеется, в этом случае могут быть утрачены некоторые или даже все преимущества параллелизма.

Но чаще всего результаты требуется накапливать в структуре данных. С этой целью можно вызвать метод toArray() и получить элементы из потока данных.

Создать обобщенный массив во время выполнения невозможно, и поэтому в результате вызова stream.toArray() возвращается массив типа Object[]. Если же требуется массив нужного типа, то этому методу следует передать конструктор такого массива, как показано ниже.

```
String[] result = stream.toArray(String[]::new);
  // В результате вызова метода stream.toArray()
  // получается массив типа Object[]
```

Для накопления элементов потока данных с другой целью имеется удобный метод `collect()`, принимающий экземпляр класса, реализующего интерфейс `Collector`. В частности, класс `Collectors` предоставляет немало фабричных методов для наиболее употребительных коллекторов. Так, для накопления потока данных в списке или множестве достаточно сделать один из следующих вызовов:

```
List<String> result = stream.collect(Collectors.toList());
```

или

```
Set<String> result = stream.collect(Collectors.toSet());
```

Если же требуется конкретная разновидность получаемого множества, то нужно сделать следующий вызов:

```
TreeSet<String> result =
        stream.collect(Collectors.toCollection(TreeSet::new));
```

Допустим, что требуется накапливать все символьные строки, сцепляя их. С целью можно сделать следующий вызов:

```
String result = stream.collect(Collectors.joining());
```

А если требуется разделитель элементов, то его можно передать методу `joining()` следующим образом:

```
String result = stream.collect(Collectors.joining(", "));
```

Если поток данных содержит объекты, отличающиеся от символьных строк, их нужно сначала преобразовать в символьные строки:

```
String result =
    stream.map(Object::toString).collect(Collectors.joining(", "));
```

Если результаты обработки потока данных требуется свести к сумме, среднему, максимуму или минимуму, воспользуйтесь методами типа `summarizing(Int|Long|Double)`. Эти методы принимают функцию, преобразующую потоковые объекты в число и возвращающую результат типа `(Int|Long|Double)SummaryStatistics`, одновременно вычисляя сумму, среднее, максимум и минимум, как показано ниже.

```
IntSummaryStatistics summary = stream.collect(
    Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

8.9. Накопление результатов в отображениях

Допустим, что имеется поток данных типа `Stream<Person>` и его элементы требуется накапливать в отображении, чтобы в дальнейшем искать людей по их идентификационному номеру. Для этой цели служит метод `Collectors.toMap()`,

принимающий в качестве двух своих аргументов функции, чтобы получить ключи и значения из отображения, как показано в следующем примере кода:

```
Map<Integer, String> idToName = people.collect(
    Collectors.toMap(Person::getId, Person::getName));
```

В общем случае, когда значения должны быть конкретными элементами, в качестве второго аргумента данному методу предоставляется функция `Function.identity()` следующим образом:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(Person::getId, Function.identity()));
```

Если же одному и тому же ключу соответствует больше одного элемента, то возникает конфликт, и коллектор генерирует исключение типа `IllegalStateException`. Это поведение можно изменить, предоставив данному методу в качестве третьего аргумента функцию, разрешающую подобный конфликт и определяющую значение по заданному ключу, исходя из существующего или нового значения. Такая функция может возвратить существующее значение, новое значение или и то и другое.

В приведенном ниже примере создается отображение, содержащее региональные настройки для каждого языка в виде ключа, обозначающего название языка в региональных настройках по умолчанию (например, `"German"`), и значения, обозначающего его локализованное название (например, `"Deutsch"`). В данном примере не учитывается, что один и тот же язык может встретиться дважды (например, немецкий в Германии и Швейцарии), и поэтому в отображении сохраняется лишь первая запись.

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
    Collectors.toMap(
        Locale::getDisplayLanguage,
        Locale::getDisplayLanguage,
        (existingValue, newValue) -> existingValue));
```

НА ЗАМЕТКУ. В этой главе в качестве структуры данных для хранения региональных настроек употребляется класс **`Locale`**. Подробнее об обращении с региональными настройками речь пойдет в главе 13.

Допустим, что требуется выяснить все языки данной страны. Для этой цели понадобится отображение типа `Map<String, Set<String>>`. Например, значением по ключу `"Switzerland"` является множество `[French, German, Italian]`. Сначала для каждого языка сохраняется одноэлементное множество. А всякий раз, когда обнаруживается новый язык заданной страны, образуется объединение из существующего и нового множеств, как показано ниже.

```
Map<String, Set<String>> countryLanguageSets = locales.collect(
    Collectors.toMap(
        Locale::getDisplayCountry,
        l -> Collections.singleton(l.getDisplayLanguage()),
        (a, b) -> { // объединить множества a и b
            Set<String> union = new HashSet<>(a);
```

```
        union.addAll(b);
        return union; }));
```

Более простой способ получения этого отображения будет представлен в следующем разделе. Если же потребуется древовидное отображение типа `TreeMap`, то в качестве четвертого аргумента методу `toMap()` следует предоставить конструктор данного класса. Необходимо также предоставить функцию объединения. Ниже приведен один из примеров из начала этого раздела, переделанный с целью получить отображение типа `TreeMap`.

```
Map<Integer, Person> idToPerson = people.collect(
  Collectors.toMap(
    Person::getId,
    Function.identity(),
    (existingValue, newValue) -> { throw new IllegalStateException(); },
    TreeMap::new));
```

НА ЗАМЕТКУ. Каждому из вариантов метода **`toMap()`** соответствует эквивалентный метод **`toConcurrentMap()`**, получающий параллельное отображение. Единое параллельное отображение применяется в процессе параллельного накопления. Если же общее отображение применяется вместе с параллельным потоком данных, то такой способ оказывается более эффективным, чем объединение множеств. Но в таком случае элементы не накапливаются в потоковом порядке, хотя это обычно не имеет особого значения.

8.10. Группирование и разделение

В предыдущем разделе было показано, как накапливаются все языки заданной страны. Но этот процесс оказался несколько трудоемким, поскольку для каждого значения из отображения пришлось сначала сформировать одноэлементное множество, а затем указать порядок объединения существующего и нового значений. Очень часто из значений с одинаковыми характеристиками образуются группы, и этот процесс непосредственно поддерживается методом `groupingBy()`.

Рассмотрим задачу группирования региональных настроек по странам. Сначала образуется следующее отображение:

```
Map<String, List<Locale>> countryToLocales = locales.collect(
    Collectors.groupingBy(Locale::getCountry));
```

Функция `Locale::getCountry()` выполняет роль *классификатора* группирования. Затем все региональные настройки можно отыскать по заданному коду страны, как показано в следующем примере кода:

```
List<Locale> swissLocales = countryToLocales.get("CH");
    // получить региональные настройки [it_CH, de_CH, fr_CH]
```

НА ЗАМЕТКУ. Как известно, все региональные настройки состоят из кода языка (например, код **`en`** обозначает английский язык) и кода страницы (например, код **`US`** обозначает Соединенные Штаты). Так, региональные настройки **`en_US`** описывают английский язык в Соединенных Штатах, а

региональные настройки **en_IE** — английский язык в Ирландии. Некоторым странам требуется несколько региональных настроек. Например, региональные настройки **ga_IE**, описывающие гэльский язык в Ирландии в дополнение к упомянутым выше региональным настройкам **en_IE**. А для Швейцарии требуются три региональные настройки, как было показано в предыдущем разделе.

Когда функция классификатора оказывается предикатной (т.е. функцией, возвращающей логическое значение типа `boolean`), элементы потока данных разделяются на основной список с элементами, для которых функция возвращает логическое значение `true`, и дополнительный список. В данном случае эффективнее воспользоваться методом `partitioningBy()`, чем методом `groupingBy()`. Так, в следующем примере кода все региональные настройки разделяются на те, что описывают английский язык, и все остальные:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(
    Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale>> englishLocales = englishAndOtherLocales.get(true);
```

НА ЗАМЕТКУ. Если вызвать метод **groupingByConcurrent()**, то в конечном итоге будет получено параллельное отображение, которое заполняется параллельно, если оно применяется вместе с параллельным потоком данных. В этом отношении данный метод очень похож на метод **toConcurrentMap()**.

8.11. Нисходящие коллекторы

Метод `groupingBy()` формирует множество, значениями которого являются списки. Если эти списки требуется каким-то образом обработать, то следует предоставить *нисходящий коллектор*. Так, если вместо списков требуются множества, то можно воспользоваться коллектором `Collectors.toSet()` следующим образом:

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(
    groupingBy(Locale::getCountry, toSet()));
```

НА ЗАМЕТКУ. В данном и последующих примерах из этого раздела предполагается статический импорт **java.util.stream.Collectors.***, чтобы упростить выражения и сделать их более удобочитаемыми.

Для сведения сгруппированных элементов к числам предоставляется ряд следующих коллекторов:

- **counting()** — производит подсчет накопленных элементов. Так, в следующем примере кода подчитывается количество региональных настроек для каждой страны:

```
Map<String, Long> countryToLocaleCounts = locales.collect(
    groupingBy(Locale::getCountry, counting()));
```

- **summing(Int|Long|Double)** — принимает в качестве аргумента функцию, применяет ее к элементам нисходящего потока данных и получает их сумму. Так, в следующем примере кода вычисляется суммарное население каждого штата из потока городов:

```
Map<String, Integer> stateToCityPopulation = cities.collect(
groupingBy(City::getState, summingInt(City::getPopulation)));
```

- **maxBy()** и **minBy()** — принимают в качестве аргумента компаратор и получают максимальный и минимальные элементы из нисходящего потока данных. Так, в следующем примере кода получается самый крупный город в каждом штате:

```
Map<String, City> stateToLargestCity = cities.collect(
    groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
```

- **mapping()** — применяет функцию к результатам, полученным из нисходящего потока данных, но для обработки результатов ему требуется еще один коллектор. Так, в следующем примере кода города группируются по штатам:

```
Map<String, Optional<String>> stateToLongestCityName = cities.collect(
    groupingBy(City::getState,
        mapping(City::getName,
            maxBy(Comparator.comparing(String::length)))));
```

- В каждом штате получаются названия городов, которые сводятся по максимальной длине.

Метод `mapping()` позволяет изящнее решить задачу из предыдущего раздела — собрать все языки, употребляемые в стране. В предыдущем разделе вместо метода `groupingBy()` применялся метод `toMap()`. А в приведенном ниже решении отпадает необходимость объединять отдельные множества.

```
Map<String, Set<String>> countryToLanguages = locales.collect(
    groupingBy(Locale::getDisplayCountry,
        mapping(Locale::getDisplayLanguage,
            toSet())));
```

Если функция группирования или отображения возвращает тип `int`, `long` или `double`, элементы можно накопить в объекте суммарной статистики, как пояснялось в разделе 8.8. Ниже показано, как это делается. А затем из объектов суммарной статистики каждой группы можно получить суммарное, подсчитанное, среднее, минимальное и максимальное значения функции.

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary =
    cities.collect(
        groupingBy(City::getState,
            summarizingInt(City::getPopulation)));
```

НА ЗАМЕТКУ. Имеются три варианта метода **reducing()**, выполняющие общие операции сведения, описываемые в следующем разделе.

Коллекторы можно эффективно сочетать вместе, но в итоге получаются весьма запутанные выражения. Поэтому их лучше всего использовать вместе с методом `groupingBy()` или `partitioningBy()` для обработки значений, преобразуемых из нисходящего потока данных. В противном случае непосредственно в потоках данных просто применяются такие методы, как `map()`, `reduce()`, `count()`, `max()` или `min()`.

8.12. Операции сведения

Метод `reduce()` реализует общий механизм для вычисления значения из потока данных. В простейшей форме он принимает двоичную функцию и применяет ее, начиная с первых двух элементов потока данных. Этот механизм проще всего пояснить на примере функции суммирования, как показано ниже.

```
List<Integer> values = ...;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

В данном примере метод `reduce()` вычисляет сумму $\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2 + \ldots$, где $\mathbf{v}_i$ — элементы потока данных. Этот метод возвращает необязательное значение типа `Optional`, поскольку достоверный результат недостижим, если поток данных пуст.

НА ЗАМЕТКУ. В данном примере можно сделать вызов **`reduce(Integer::sum)`** вместо вызова **`reduce((x, y) -> x + y)`**.

В общем, если метод `reduce()` выполняет операцию сведения ***op***, то она дает результат $\mathbf{v}_0$ ***op*** $\mathbf{v}_1$ ***op*** $\mathbf{v}_2$ ***op*** $\ldots$, где $\mathbf{v}_i$ ***op*** $\mathbf{v}_{i+1}$ обозначает вызов функции ***op***$(\mathbf{v}_i, \mathbf{v}_{i+1})$. Эта операция должна быть *ассоциативной*, т.е. порядок сочетания ее элементов значения не имеет. В математическом обозначении операция $(x\ op\ y)\ op\ z$ должна быть равнозначна операции $x\ op\ (y\ op\ z)$. Это дает возможность выполнять эффективное сведение в параллельных потоках данных.

Практическую пользу могут принести многие ассоциативные операции, в том числе сложение, умножение, сцепление символьных строк, получение максимума и минимума, объединение и пересечение множеств. Примером операции, которая не является ассоциативной, служит вычитание. Так, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Нередко имеется *тождественный элемент* ***e*** вроде ***e op x = x***, и он может быть использован в качестве отправной точки для вычисления. Например, `0` является тождественным элементом операции сложения. Ниже приведена вторая форма вызова метода `reduce()`. Тождественное значение возвращается в том случае, если поток данных пуст и больше не нужно обращаться к классу `Optional`.

```
List<Integer> values = ...;
Integer sum = values.stream().reduce(0, (x, y) -> x + y)
  // Вычисляет результат 0 + v0 + v1 + v2 + . . .
```

А теперь допустим, что имеется поток объектов и требуется получить сумму некоторых свойств, например, длину всех символьных строк в потоке. Для этой цели не годится простая форма метода `reduce()`, поскольку в ней требуется функция `(T, T) -> T` с одинаковыми типами аргументов и возвращаемого результата. Но в данном

случае имеются два разных типа: `String` — для элементов потока данных и `int` — для накапливаемого результата. На этот случай имеется отдельная форма вызова метода `reduce()`.

Прежде всего нужно предоставить функцию накопления `(total, word) -> total + word.length()`. Эта функция вызывается повторно, образуя сумму нарастающим итогом. Но если вычисление этой суммы распараллелено, то оно разделяется на несколько параллельных вычислений, результаты которых должны быть объединены. Для этой цели предоставляется вторая функция. Ниже приведена полная форма вызова метода `reduce()` в данном случае.

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```

НА ЗАМЕТКУ. На практике методом **`reduce()`** приходится пользоваться нечасто. Ведь намного проще преобразовать исходный поток символьных строк в поток чисел и воспользоваться одним из методов для вычисления суммы, максимума или минимума. (Подробнее потоки чисел рассматриваются далее, в разделе 8.13.) В данном конкретном случае можно было бы сделать вызов **`words.mapToInt(String::length).sum()`**. Это было бы проще и эффективнее, поскольку не потребовало бы упаковки.

НА ЗАМЕТКУ. Иногда метод **`reduce()`** оказывается недостаточно обобщенным. Допустим, что требуется накопить результаты в множестве типа **`BitSet`**. Если распараллелить эту коллекцию, то разместить ее элементы в одном множестве типа **`BitSet`** не удастся, поскольку объект типа **`BitSet`** не является потокобезопасным. Именно поэтому нельзя воспользоваться методом **`reduce()`**. Каждый сегмент исходной коллекции должен начинаться со своего пустого множества, а методу **`reduce()`** можно предоставить только одно тождественное значение. В таком случае следует воспользоваться методом **`collect()`**, который принимает следующие аргументы.

1. *Поставщик* для получения новых экземпляров целевого объекта. Например, конструктор для построения хеш-множества.
2. *Накопитель*, вводящий элемент в целевой объект. Например, метод **`add()`**.
3. *Объединитель*, соединяющий два объекта в один. Например, метод **`addAll()`**.

Ниже показано, каким образом метод **`collect()`** вызывается для множества битов.

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

8.13. Потоки данных примитивных типов

До сих пор целочисленные значения накапливались в потоке данных типа `Stream<Integer>`, несмотря на то, что заключать каждое целочисленное значение в объект-оболочку совершенно неэффективно. Это же относится и к другим примитивным типам данных `double`, `float`, `long`, `short`, `char`, `byte` и `boolean`. В библиотеке потоков данных имеются специализированные классы `IntStream`, `LongStream` и `DoubleStream`, позволяющие сохранять значения примитивных типов непосредственно, не прибегая к помощи оболочек. Так, если требуется сохранить значения типа

short, char, byte и boolean, достаточно воспользоваться классом IntStream, а для хранения значений типа float — классом DoubleStream.

Чтобы создать поток данных типа IntStream, достаточно вызвать методы IntStream.of() и Arrays.stream() следующим образом:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to);
    // массив values относится к типу int[]
```

К потокам данных примитивных типов, как и к потокам объектов, можно применять статические методы generate() и iterate(). Кроме того, в классах IntStream и LongStream имеются статические методы range() и rangeClosed(), генерирующие диапазоны целочисленных значений с единичным шагом, как показано ниже.

```
IntStream zeroToNinetyNine = IntStream.range(0, 100);
    // Верхний предел исключительно
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
    // Верхний предел включительно
```

В интерфейсе CharSequence имеются методы codePoints() и chars(), получающие поток типа IntStream кодов символов в Юникоде или кодовых единиц в кодировке UTF-16. (Подробнее о кодировках символов см. в главе 1.) Ниже приведен пример применения метода codePoints().

```
String sentence = "\uD835\uDD46 is the set of octonions.";
    // \uD835\uDD46 — это кодировка UTF-16 знака 𝕆, обозначающего
    // октонионы в Юникоде (U+1D546)

IntStream codes = sentence.codePoints();
    // Поток шестнадцатеричных значений 1D546 20 69 73 20 . . .
```

Поток объектов можно преобразовать в поток данных примитивных типов с помощью методов mapToInt(), mapToLong(() или mapToDouble(). Так, если имеется поток символьных строк и их длины требуется обработать как целочисленные значения, это можно сделать и средствами класса IntStream следующим образом:

```
Stream<String> words = ...;
IntStream lengths = words.mapToInt(String::length);
```

Чтобы преобразовать поток данных примитивного типа в поток объектов, достаточно воспользоваться методом boxed() следующим образом:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Как правило, методы для потоков данных примитивных типов аналогичны методам для потоков объектов. Ниже перечислены наиболее существенные их отличия.

- Методы типа toArray возвращают массивы примитивных типов.
- Методы, возвращающие результат необязательного типа, возвращают значение типа OptionalInt, OptionalLong или OptionalDouble. Классы этих типов аналогичны классу Optional, но у них имеются методы getAsInt(), getAsLong() и getAsDouble() вместо метода get().

- Имеются методы `sum()`, `average()`, `max()` и `min()`, возвращающие сумму, среднее, максимум и минимум соответственно. Эти методы не определены для потоков объектов.
- Метод `summaryStatistics()` возвращает объект типа `IntSummaryStatistics`, `LongSummaryStatistics` или `DoubleSummaryStatistics`, способный одновременно сообщать о сумме, среднем, максимуме и минимуме в потоке данных.

НА ЗАМЕТКУ. В классе **`Random`** имеются методы **`ints()`**, **`longs()`** и **`doubles()`**, возвращающие потоки данных примитивных типов, состоящие из случайных чисел.

8.14. Параллельные потоки данных

Потоки данных упрощают распараллеливание групповых операций. Этот процесс происходит в основном автоматически, но требует соблюдения немногих правил. Прежде всего, нужно иметь в своем распоряжении параллельный поток данных. Получить параллельный поток данных можно из любой коллекции с помощью метода `Collection.parallelStream()` следующим образом:

```
Stream<String> parallelWords = words.parallelStream();
```

Более того, метод `parallel()` преобразует любой последовательный поток данных в параллельный, как показано ниже. При выполнении оконечного метода поток данных действует в параллельном режиме, и поэтому промежуточные операции в этом потоке распараллеливаются.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

Если потоковые операции выполняются параллельно, то и получаемый в итоге результат должен быть таким же, как и при последовательном выполнении операций. Очень важно, чтобы эти операции выполнялись *без сохранения состояния* и в произвольном порядке.

Допустим, что требуется подсчитать все короткие слова в потоке символьных строк. В приведенном ниже примере демонстрируется, как не следует решать эту задачу.

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
      // ОШИБКА: состояние гонок!
System.out.println(Arrays.toString(shortWords));
```

Приведенный выше код написан очень скверно. Функция, передаваемая методу `forEach()`, выполняется параллельно в нескольких потоках исполнения, в каждом из которых обновляется разделяемый ими общий массив. Как будет показано в главе 10, это классическое *состояние гонок*. Если выполнить данный код многократно, то в

результате каждого его выполнения, вероятнее всего, будет получена совсем другая последовательность подсчитанных коротких слов, причем каждый раз неверная.

В обязанности программиста входит обеспечение надежного выполнения в параллельном режиме функций, передаваемых для распараллеливания операций в потоке данных. Для этого лучше всего избегать изменяемого состояния. В следующем примере кода вычисления можно надежно распараллелить, если сгруппировать символьные строки по длине и подсчитать их:

```
Map<Integer, Long> shortWordCounts =
    words.parallelStream()
        .filter(s -> s.length() < 10)
        .collect(groupingBy(
            String::length,
            counting()));
```

По умолчанию потоки данных, получаемые из упорядоченных коллекций (массивов и списков), диапазонов, генераторов, итераторов или в результате вызова метода `Stream.sorted()`, *упорядочиваются*. Результаты накапливаются в порядке следования исходных элементов и полностью предсказуемы. Если выполнить одни и те же операции дважды, то будут получены совершенно одинаковые результаты.

Упорядочение не исключает эффективное распараллеливание. Например, при вызове `stream.map(fun)` поток данных может быть разбит на *n* сегментов, каждый из которых обрабатывается параллельно. А полученные результаты снова собираются по порядку.

Некоторые операции могут быть распараллелены более эффективно, если требование упорядочения опускается. Вызывая метод `Stream.unordered()`, можно указать, что упорядочение не имеет значения. Это, в частности, выгодно при выполнении операции методом `Stream.distinct()`. В упорядоченном потоке метод `distinct()` сохраняет первый из всех равных элементов. Этим ускоряется распараллеливание, поскольку в потоке исполнения, обрабатывающем отдельный сегмент, неизвестно, какие именно элементы следует отбросить, до тех пор, пока сегмент не будет обработан. Если же допускается сохранить *любой* однозначный элемент, то все сегменты могут быть обработаны параллельно (с помощью общего множества для отслеживания дубликатов).

Если опустить упорядочение, то можно также ускорить выполнение метода `limit()`. Если же требуется обработать *любые* *n* элементов из потока данных и при этом неважно, какие из них будут получены, то с этой целью можно сделать следующий вызов:

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

Как обсуждалось в разделе 8.9, объединять отображения невыгодно. Именно поэтому в методе `Collectors.groupingByConcurrent()` используется общее параллельное отображение. Чтобы извлечь выгоду из параллелизма, порядок следования значений в отображении должен быть иным, чем в потоке данных:

```
Map<Integer, List<String>> result = words.parallelStream().collect(
    Collectors.groupingByConcurrent(String::length));
    // Значения не накапливаются в потоковом порядке
```

Разумеется, это не имеет особого значения, если применяется нисходящий коллектор, не зависящий от упорядочения, как в следующем примере кода:

```
Map<Integer, Long> wordCounts =
    words.parallelStream()
        .collect(
            groupingByConcurrent(
                String::length,
                counting()));
```

ВНИМАНИЕ. При выполнении потоковой операции очень важно не изменять коллекцию, поддерживающую поток данных, даже если такое изменение и является потокобезопасным. Напомним, что данные в потоках не накапливаются, а всегда находятся в отдельной коллекции. Если попытаться изменить коллекцию, то результат выполнения потоковых операций окажется неопределенным. В документации на комплект JDK такое требование называется *невмешательством*. Оно относится как к последовательным, так и к параллельным потокам данных.

Точнее говоря, коллекцию можно изменять вплоть до момента выполнения оконечной операции, поскольку промежуточные потоковые операции выполняются по требованию. Так, следующий фрагмент кода вполне работоспособен, хотя и не рекомендуется:

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

А приведенный ниже фрагмент кода оказывается неработоспособным.

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
     // ОШИБКА: вмешательство!
```

Упражнения

1. Убедитесь, что запрос первых пяти длинных слов не требует вызова метода `filter()`, если найдено пятое длинное слово. С этой целью просто организуйте протоколирование вызова каждого метода.
2. Определите разность во времени подсчета длинных слов с помощью методов `parallelStream()` и `stream()`. Вызовите метод `System.currentTimeMillis()` до и после этих методов и выведите разность. Если у вас быстродействующий компьютер, выберите для подсчета длинных слов более длинный документ (например, роман "Война и мир").
3. Допустим, что имеется массив `int[] values = { 1, 4, 9, 16 }`. Каков результат вызова `Stream.of(values)`? Как вместо этого получить поток данных типа `int`?
4. С помощью метода `Stream.iterate()` создайте бесконечный поток случайных чисел, не вызывая метод `Math.random()`, а непосредственно реализуя *линейный конгруэнтный генератор*. Такой генератор начинает действовать с числа, задаваемого выражением x_0 = *начальное значение*, а затем производит случайные

числа по формуле $x_{n+1} = (a\ x_n + c)\ \%\ m$, при соответствующих значениях a, c и m. С этой целью реализуйте метод, принимающий параметры a, c, m и получающий поток данных `Stream<Long>`. Опробуйте его со следующими параметрами: a = **25214903917**, c = **11** и $m = 2^{48}$.

5. Метод `letters()`, упоминавшийся в разделе 8.3, выглядит несколько неуклюже, поскольку в нем сначала получается списочный массив, а затем он превращается в поток данных. Напишите другой, однострочный вариант этого метода, используя поток данных. Преобразуйте значения типа `int` в пределах от **0** до `s.length() - 1` с помощью подходящего лямбда-выражения.
6. Воспользуйтесь методом `String.codePoints()` для реализации метода, проверяющего, является ли символьная строка словом, состоящим только из букв. (*Подсказка:* воспользуйтесь методом `Character.isAlphabetic()`.) Реализуйте тем же самым способом метод, проверяющий, является ли символьная строка достоверным в Java идентификатором.
7. Преобразовав содержимое файла в поток лексем, выведите список первых 100 лексем, являющихся словами в том смысле, в каком они определены в предыдущем упражнении. Прочитайте содержимое файла снова и выведите список из 10 наиболее часто употребляемых слов, игнорируя регистр букв.
8. Прочитайте слова из файла `/usr/share/dict/words` (или аналогичного списка слов) в поток данных и получите массив всех слов, содержащих пять отдельных гласных.
9. Определите среднюю длину строки в заданном конечном потоке символьных строк.
10. Определите все символьные строки максимальной длины в заданном конечном потоке символьных строк.
11. Допустим, что ваш непосредственный начальник дал вам задание написать метод `public static <T> boolean isFinite(Stream<T> stream)`. Почему это не самая удачная мысль? Все равно напишите этот метод.
12. Напишите метод `public static <T> Stream<T> zip(Stream<T> first, Stream<T> second)`, изменяющий элементы из потоков данных `first` и `second` (или возвращающий пустое значение `null`, если в потоке данных, черед которого настает, исчерпываются элементы).
13. Соедините все элементы в потоках данных `Stream<ArrayList<T>>` и `ArrayList<T>`. Покажите, как добиться этого с помощью каждой из трех форм метода `reduce()`.
14. Организуйте вызов метода `reduce()` таким образом, чтобы вычислить среднее в потоке данных `Stream<Double>`. Почему нельзя просто вычислить сумму и разделить ее на результат, возвращаемый методом `count()`?
15. Найдите 500 простых чисел с 50 десятичными цифрами, используя параллельный поток данных типа `BigInteger` и метод `BigInter.isProbablePrime()`.

Насколько это делается быстрее, чем при использовании последовательного потока данных?

16. Найдите 500 самых длинных слов в романе "Война и мир", используя параллельный поток данных. Насколько это делается быстрее, чем при использовании последовательного потока данных?
17. Каким образом можно исключить получение смежных дубликатов из потока данных? Сможет ли написанный вами метод обрабатывать параллельный поток?

Глава 9

Организация ввода-вывода

В этой главе...

- 9.1. Потоки ввода, вывода, чтения и записи
- 9.2. Каталоги, файлы и пути к ним
- 9.5. Подключения по заданному URL
- 9.4. Регулярные выражения
- 9.5. Сериализация
- Упражнения

В этой главе поясняется, как обращаться с файлами, словарями и веб-страницами и как вводить и выводить данные в двоичном и текстовом формате. В ней также рассматриваются регулярные выражения, удобные для обработки вводимых данных. (Это самое подходящее место для обсуждения данной темы, поскольку прикладной программный интерфейс API для регулярных выражений был внедрен в спецификацию Java по запросу новых средств ввода-вывода.) И наконец, в этой главе поясняется механизм сериализации объектов, позволяющий хранить объекты так же просто, как и текст или числовые данные.

Основные положения этой главы приведены ниже.

1. Потоки ввода служат источниками байтов, а потоки вывода — адресатами байтов.
2. Потоки чтения и записи служат для обработки символов. Для этого следует непременно указывать кодировку символов.
3. В классе `Files` имеются служебные методы для чтения всех байтов или строк из файла.
4. В интерфейсах `DataInput` и `DataOutput` имеются методы для вывода чисел в двоичном формате.
5. Для произвольного доступа служит класс `RandomAccessFile` или отображаемый в памяти файл.
6. Класс `Path` представляет составляющие абсолютного или относительного пути к файлам в файловой системе. Пути могут быть объединены (иными словами, "разрешены").
7. Методы из класса `Files` служат для копирования, перемещения или удаления файлов и рекурсивного обхода дерева каталогов.
8. Для чтения или обновления архивного ZIP-файла служит система файлов формата ZIP.
9. Для чтения содержимого веб-страницы служит класс `URL`, а для чтения или записи метаданных — класс `URLConnection`.
10. С помощью классов `Pattern` и `Matcher` можно обнаружить все совпадения с регулярным выражением в символьной строке, а для каждого совпадения — фиксированные группы.
11. Механизм сериализации позволяет сохранять и восстанавливать объект любого класса, реализующего интерфейс `Serializable`, при условии, что переменные экземпляра этого объекта также сериализируются.

9.1. Потоки ввода, вывода, чтения и записи

В прикладном программном интерфейсе Java API источник, из которого вводятся байты, называется *потоком ввода*. Байты могут поступать в оперативную память из файла, через сетевое соединение или из массива. Аналогично адресатом байтов служит *поток вывода*. С другой стороны, *потоки чтения* и *записи* потребляют и

поставляют последовательности *символов*. (Все эти разновидности потоков ввода-вывода не имеют никакого отношения к потокам данных, рассматривавшимся в главе 8.) В последующих разделах поясняется, как вводить и выводить байты и символы.

9.1.1. Получение потоков ввода-вывода

Получить поток ввода-вывода из файла проще всего с помощью следующих статических методов:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

В данном примере аргумент `path` обозначает экземпляр класса `Path`, рассматриваемого далее, в разделе 9.2.1. Этот класс описывает путь к файлу в файловой системе.

Если же имеется URL, то прочитать содержимое ресурса по этому URL можно из потока ввода, возвращаемого методом `openStream()` из класса `URL`, как показано ниже. В разделе 9.3 далее в этой главе поясняется, как записывать данные по указанному URL.

```
URL url = new URL("http://horstmann.com/index.html");
InputStream in = url.openStream();
```

Класс `ByteArrayInputStream` позволяет читать данные из массива байтов следующим образом:

```
byte[] bytes = ...;
InputStream in = new ByteArrayInputStream(bytes);
```

С другой стороны, для вывода данных в массива байтов служит класс `ByteArrayOutputStream`, как показано ниже.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
Направить байты в стандартный поток вывода
byte[] bytes = out.toByteArray();
```

9.1.2. Ввод байтов

В классе `InputStream` имеется приведенный ниже метод для ввода одного байта. Этот метод возвращает байт в виде целочисленного значения в пределах от **0** до **255**, а по достижении конца вводимых данных — значение **-1**.

```
InputStream in = ...;
int b = in.read();
```

ВНИМАНИЕ. Тип **byte** в Java определяет целочисленные значения в пределах от **-128** до **127**. Тип возвращаемого значения можно привести к типу **byte** *после* того, как только будет проверено, что это не значение **-1**.

Чаще всего байты требуется вводить не по одному, а массово. Для размещения байтов из потока ввода в массиве служат два приведенных ниже варианта метода `read()`. Оба эти варианта данного метода вводят байты до тех пор, пока не заполнится

массив, указанный диапазон или же вводить больше нечего, а возвращают они конкретное число введенных байтов. Если вводимые данные вообще отсутствуют, то оба метода возвращают значение -1.

```
byte[] bytes = ...;
actualBytesRead = in.read(bytes);
actualBytesRead = in.read(bytes, start, length);
```

В библиотеке Java отсутствует метод для чтения всех байтов из потока ввода. Ниже демонстрируется один из способов сделать это в прикладном коде. Описание вспомогательного метода `copy()` приведено в следующем разделе.

```
public static byte[] readAllBytes(InputStream in) throws IOException {
ByteArrayOutputStream out = new ByteArrayOutputStream();
copy(in, out);
out.close();
return out.toByteArray();
}
```

СОВЕТ. Если из файла требуется прочитать все байты, то рекомендуется вызвать следующий вспомогательный метод:

```
byte[] bytes = Files.readAllBytes(path);
```

9.1.3. Вывод байтов

Следующие формы метода `write()` из класса `OutputStream` позволяют выводить байты как по отдельности, так и целыми массивами:

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

По окончании вывода данных в поток последний нужно закрыть, чтобы зафиксировать любые буферизованные для вывода данные. Это лучше всего сделать с помощью оператора `try` с ресурсами следующим образом:

```
try (OutputStream out = ...) {
  out.write(bytes);
}
```

Если поток ввода требуется скопировать в поток вывода, то для этой цели служит следующий вспомогательный метод:

```
public static void copy(InputStream in, OutputStream out) throws IOException {
  final int BLOCKSIZE = 1024;
  byte[] bytes = new byte[BLOCKSIZE];
  int len;
  while ((len = in.read(bytes)) != -1) out.write(bytes, 0, len);
}
```

А для сохранения потока ввода типа `InputStream` в файле достаточно сделать следующий вызов:

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

9.1.4. Кодировки символов

Потоки ввода-вывода представляют собой последовательности байтов, но зачастую приходится обрабатывать текст, т.е. последовательности символов. В таком случае имеет значение, каким образом символы кодируются в байты.

Для кодирования символов в Java применяется стандарт, называемый *Юникодом*. Каждый символ, или так называемая *кодовая точка*, представлен в Юникоде 21-разрядным целым числом. Имеются разные *кодировки символов* — способы упаковки 21-разрядных целых чисел в байты.

Чаще всего применяется кодировка UTF-8, в которой каждая кодовая точка в Юникоде кодируется последовательностью от одного до четырех байтов (табл. 9.1). Преимущество кодировки UTF-8 состоит в том, что символы из традиционного набора в коде ASCII, куда входят все буквы латинского и английского алфавита, занимают только один байт.

Таблица 9.1. Кодировка UTF-8

Диапазон символов	Кодировка
`0...7F`	$0a_6a_5a_4a_3a_2a_1a_0$
`80...7FF`	$110a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
`800...FFFF`	$1110a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$11110a_{20}a_{19}a_{18}\ 10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$

Еще одной распространенной является кодировка UTF-16, в которой каждая кодовая точка в Юникоде кодируется одним или двумя 16-разрядными значениями (табл. 9.2). Такая кодировка применяется в символьных строках Java. На самом деле имеются две формы кодировки UTF-16: с обратным порядком байтов и с прямым порядком следования байтов. Рассмотрим в качестве примера 16-разрядное значение `0x2122`. В формате с обратным порядком байтов первым следует старший байт `0x21`, а за ним — младший байт `0x22`. А в формате с прямым порядком байтов все происходит наоборот: сначала следует младший байт `0x22`, а затем старший байт `0x21`. Для обозначения используемого порядка следования байтов файл может начинаться с соответствующей метки в виде 16-разрядного значения `0xFEFF`. С помощью этой метки пользователь может определить порядок следования байтов , после чего отбросить ее.

Таблица 9.2. Кодировка UTF-16

Диапазон символов	Кодировка
`0...FFFF`	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}\ 110111a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$, где $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

ВНИМАНИЕ. В некоторых прикладных программах, в том числе в текстовом редакторе Microsoft Notepad, метка порядка следования байтов вводится в начале файлов, содержимое которых представлено в кодировке UTF-8. Очевидно, что это излишне, поскольку в кодировке UTF-8 вопросы, связанные с порядком следования байтов, не возникают. Тем не менее стандарт на Юникод допускает наличие такой метки и даже считает это целесообразным, поскольку она разрешает всякие сомнения по поводу кодировки. При чтении содержимого файла в кодировке UTF-8 такая метка должна удаляться. К сожалению, в Java этого не делается, а в отчетах об устраненных программных ошибках напротив данной ошибки стоит метка "не устранимая". Поэтому любой начальный код \ **uFEFF**, обнаруживаемый при вводе данных, придется удалять вручную.

Помимо упомянутых выше кодировок UTF, имеются частичные кодировки, охватывающие диапазон символов, пригодный для конкретного круга пользователей. Например, кодировка по стандарту ISO 8859-1 определяет однобайтовый код, включающий в себя символы с ударениями, применяемые в западноевропейских языках, а кодировка Shift-JIS — код переменной длины для японских символов. Немалое число этих кодировок по-прежнему широко распространено.

Надежного способа выявлять кодировку символов в потоке ввода байтов не существует. В некоторых методах из прикладного программного интерфейса API допускается применение "набора символов по умолчанию" — кодировки символов, которая считается наиболее предпочтительной в операционной системе компьютера. Но применяется ли та же самая кодировка и в источнике байтов? Ведь эти байты вполне могут поступать из разных частей света. Следовательно, кодировка символов должна всегда указываться явно. Так, при чтении веб-страницы следует проверять заголовок `Content-Type`.

НА ЗАМЕТКУ. Кодировка, принятая на конкретной платформе, возвращается статическим методом **`Charset.defaultCharset()`**. А статический метод **`Charset.availableCharsets()`** возвращает все имеющиеся экземпляры класса **`Charset`**, преобразованные из канонических имен в объекты типа **`Charset`**.

ВНИМАНИЕ. В реализации библиотеки Java от компании Oracle имеется системное свойство **`file.encoding`** для переопределения кодировки символов, принятой на конкретной платформе по умолчанию. Но это свойство не поддерживается официально и не последовательно согласуется со всеми частями реализации библиотеки Java от компании Oracle. Поэтому устанавливать его не следует.

В классе `StandardCharsets` имеются следующие статические переменные типа `Charset` для тех кодировок, которые должны поддерживаться на каждой виртуальной машине Java:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

Чтобы получить объект типа Charset для другой кодировки, достаточно вызвать статический метод forName() следующим образом:

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

Для ввода-вывода текста следует пользоваться объектом типа Charset. В следующем примере показано, как превратить массив байтов в символьную строку:

```
String str = new String(bytes, StandardCharsets.UTF_8);
```

НА ЗАМЕТКУ. При вызове некоторых методов допускается указывать кодировку символов с помощью объекта типа **Charset** или символьной строки. Чтобы не утруждать себя указанием правильного написания кодировки, достаточно выбрать подходящую константу из класса **StandardCharsets**. Так, выражение **new String(bytes, "UTF 8")** неприемлемо и вызовет ошибку.

ВНИМАНИЕ. В одних методах и конструкторах, например, в конструкторе **String(byte[])**, используется кодировка, принятая на платформе по умолчанию, если не указано иное. А в других методах и конструкторах, например, в конструкторе **Files.readAllLines()**, применяется кодировка UTF-8.

9.1.5. Ввод текста

Для ввода текста служит класс Reader, представляющий поток чтения. Этот поток можно получить из потока ввода с помощью класса адаптера InputStreamReader следующим образом:

```
InputStream inStream = ...;
Reader in = new InputStreamReader(inStream, charset);
```

Если требуется обработать вводимые данные по отдельным кодовым единицам в кодировке UTF-16, то для этой цели можно вызвать метод read(), как показано ниже. Этот метод возвращает кодовую единицу в пределах от **0** до **65536**, а по завершении ввода — значение **-1**.

```
int ch = in.read();
```

Но это не очень удобно, и поэтому имеется ряд других вариантов ввода текста. Так, короткий текст можно прочитать в символьную строку следующим образом:

```
String content = new String(Files.readAllBytes(path), charset);
```

Но если содержимое файла требуется ввести в виде последовательности строк, то для этой цели достаточно сделать следующий вызов:

```
List<String> lines = Files.readAllLines(path, charset);
```

А еще лучше обработать эти строки по требованию в виде потока данных типа Stream, как показано ниже.

```
try (Stream<String> lines = Files.lines(path, charset)) {
  ...
}
```

НА ЗАМЕТКУ. Если при извлечении строк в поток данных возникает исключение типа `IOException`, то оно заключается в оболочку исключения типа `UncheckedIOException`, которое генерируется в потоковой операции. (Такой прием требуется потому, что потоковые операции не объявляются для генерирования проверяемых исключений.)

Для чтения чисел или слов из файла служит класс `Scanner`, как пояснялось в главе 1. Ниже приведен характерный пример его применения в этих целях.

```
Scanner in = new Scanner(path, "UTF-8");
while (in.hasNextDouble()) {
  double value = in.nextDouble();
  ...
}
```

СОВЕТ. Чтобы прочитать слова в алфавитном порядке, следует задать разделитель для потока сканирования в регулярном выражении, дополняющем то, что требуется принять в виде маркера. Например, после следующего вызова:

```
in.useDelimiter("\\PL+");
```

в потоке сканирования будут прочитаны буквы, поскольку любая последовательность небуквенных символов считается разделителем. Подробнее о синтаксисе регулярных выражений речь пойдет в разделе 9.4.1.

Если вводимые данные поступают не из файла, поток ввода типа `InputStream` следует заключить в оболочку буферизируемого потока чтения типа `BufferedReader`, как показано ниже.

```
try (BufferedReader reader =
      new BufferedReader(new InputStreamReader(url.openStream()))) {
  Stream<String> lines = reader.lines();
  ...
}
```

Поток чтения типа `BufferedReader` читает вводимые данные по частям ради большей эффективности. (Как ни странно, элементарные потоки чтения не работают в этом режиме.) В классе `BufferedReader` имеется метод `readLine()` для чтения одной строки и метод `lines()` для получения потока строк. Если же для чтения данных из файла требуется объект типа `Reader`, то достаточно сделать следующий вызов:

```
Files.newBufferedReader(path, charset).
```

9.1.6. Вывод текста

Для вывода текста служит класс `Writer`, представляющий поток записи. Так, с помощью метода `write()` можно вывести символьные строки. Превратить поток вывода в поток записи типа `Writer` можно следующим образом:

```
OutputStream outStream = ...;
Writer out = new OutputStreamWriter(outStream, charset);
out.write(str);
```

А получить поток записи данных в файл можно таким образом:

```
Writer out = Files.newBufferedWriter(path, charset);
```

Для вывода текста намного удобнее пользоваться классом `PrintWriter`, где имеются методы `print()`, `println()` и `printf()`, которые всегда применялись вместе со стандартным потоком вывода `System.out`. С помощью этих методов можно выводить числа и отформатированные данные. Для вывода данных в файл достаточно построить объект типа `PrintWriter` следующим образом:

```
PrintWriter out = new PrintWriter(Files.newBufferedWriter(path, charset));
```

А для вывода данных в другой поток служит приведенный ниже конструктор. Обратите внимание на то, что этому конструктору класса `PrintWriter` в качестве аргумента нужно предоставить кодировку в виде символьной строки, а не объект типа `Charset`.

```
PrintWriter out = new PrintWriter(outStream, "UTF-8");
```

НА ЗАМЕТКУ. Стандартный поток вывода **`System.out`** является экземпляром класса **`PrintStream`**, а не **`PrintWriter`**. Этот пережиток остался от ранней стадии развития Java. Но методы **`print()`**, **`println()`** и **`printf()`** действуют одинаково для объектов обоих классов, **`PrintStream`** и **`PrintWriter`**, используя кодировку символов для преобразования символов в байты.

Если текст уже подготовлен для вывода в символьную строку, достаточно сделать следующий вызов:

```
String content = ...;
Files.write(path, content.getBytes(charset));
```

или такой вызов:

```
Files.write(path, lines, charset);
```

В последнем примере в качестве аргумента `lines` может быть указана ссылка на интерфейс `Collection<String>`, а в еще более общем случае — ссылка на интерфейс `Iterable<? Extends CharSequence>`. Для присоединения выводимых данных к файлу достаточно следующих строк кода:

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
Files.write(path, lines, charset, StandardOpenOption.APPEND);
```

ВНИМАНИЕ. При выводе текста с частичным набором символов вроде кодировки по стандарту ISO 8859-1 любые непреобразуемые символы негласно заменяются, как правило, знаком ? или знаком замены (**U+FFFD**) в Юникоде.

Иногда для вывода данных библиотечному методу требуется объект типа Writer. Если же выводимые данные требуется зафиксировать в символьной строке, то данному методу следует передать объект типа StringWriter. А если требуется объект типа PrintWriter, то его следует заключить в оболочку типа StringWriter:

```
StringWriter writer = new StringWriter();
throwable.printStackTrace(new PrintWriter(writer));
String stackTrace = writer.toString();
```

9.1.7. Ввод-вывод двоичных данных

В интерфейсе DataInput объявляются перечисленные ниже методы типа **read**, предназначенные для ввода чисел, символов, логических значений типа boolean или символьных строк в двоичном формате. А в интерфейсе DataOutput объявляются соответствующие методы **write** для вывода аналогичных данных в том же формате.

```
byte readByte()
int readUnsignedByte()
char readChar()
short readShort()
int readUnsignedShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
void readFully(byte[] b)
```

НА ЗАМЕТКУ. Эти методы вводят и выводят числа в формате с обратным порядком следования байтов.

ВНИМАНИЕ. Имеются также методы типа **readUTF/writeUTF**, в которых применяется формат так называемой "измененной кодировки UTF-8" (Modified UTF-8). Эти методы *несовместимы* с обычной кодировкой UTF-8 и удобны только для внутреннего механизма виртуальной машины JVM.

Преимущества ввода-вывода данных в двоичном формате заключается в том, что он имеет фиксированную длину и эффективен. Например, метод writeInt() всегда выводит целочисленное значение в виде 4 байтов, следующих в обратном порядке, т.е. от старшего к младшему, независимо от количества цифр. Для каждого значения заданного типа требуется одно и то же место, благодаря чему ускоряется произвольный доступ к данным. Кроме того, двоичные данные вводятся быстрее, чем текст, требующий синтаксического анализа. А главный недостаток ввода-вывода данных в двоичном формате заключается в том, что получающиеся в итоге файлы нельзя так просто просмотреть с текстовом редакторе.

Вместе с любым потоком ввода-вывода могут быть использованы классы адаптеров DataInputStream и DataOutputStream, как показано в следующем пример кода:

```
DataInput in = new DataInputStream(Files.newInputStream(path));
DataOutput out = new DataOutputStream(Files.newOutputStream(path));
```

9.1.8. Произвольный доступ к файлам

Класс RandomAccessFile позволяет вводить и выводить данные в любом месте файла. В частности, файл можно открыть в режиме произвольного доступа как для чтения и записи, так и только для чтения. С этой целью в качестве второго аргумента конструктору данного класса указывается символьная строка "r" (доступ только для чтения) или "rw" (доступ для чтения и записи), как показано в следующей строке кода:

```
RandomAccessFile file = new RandomAccessFile(path.toString(), "rw");
```

Для произвольного доступа к файлу имеется *указатель файла*, обозначающий позицию следующего вводимого или выводимого байта. С помощью метода seek() указатель файла устанавливается на произвольной позиции байта в файле. В качестве аргумента методу seek() указывается целочисленное значение в пределах от нуля до длины файла, которую можно получить, вызвав метод length(). А метод getFilePointer() возвращает текущую позицию указателя файла.

Класс RandomAccessFile реализует оба интерфейса — DataInput и DataOutput. Для ввода-вывода чисел из файла с произвольным доступом служат методы вроде readInt() и writeInt(), упоминавшиеся в предыдущем разделе. В следующем примере демонстрируется их применение.

```
int value = file.readInt();
file.seek(file.getFilePointer() - 4);
file.writeInt(value + 1);
```

9.1.9. Отображаемые в памяти файлы

Отображаемые в памяти файлы обеспечивают другой, весьма эффективный подход к произвольному доступу, вполне пригодный для обращения с крупными файлами. Но прикладной программный интерфейс API для доступа к данным совершенно отличается от потоков ввода-вывода. С этой целью для файла сначала получается *канал* следующим образом:

```
FileChannel channel = FileChannel.open(path,
    StandardOpenOption.READ, StandardOpenOption.WRITE)
```

Затем часть файла, а если он не слишком крупный, то и весь файл, отображается в оперативную память следующим образом:

```
ByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE,
    0, channel.size());
```

Для ввода значений служат методы get(), getInt(), getDouble() и т.д., а для вывода значений — соответствующие методы типа **put**. В какой-то момент, когда канал безусловно закрыт, эти изменения записываются обратно в файл, как показано ниже.

```
int offset = ...;
int value = buffer.getInt(offset);
buffer.put(offset, value + 1);
```

НА ЗАМЕТКУ. По умолчанию в методах для ввода-вывода чисел используется обратный порядок следования байтов. Этот порядок можно изменить по такой команде:

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

9.1.10. Блокировка файлов

Если несколько одновременно выполняющихся программ изменяют один и тот же файл, им нужно каким-то образом взаимодействовать вместе, иначе файл может быть легко разрушен. Из этого положения позволяет выйти блокировка файлов.

Допустим, что в прикладной программе сохраняется файл конфигурации с глобальными параметрами пользователя. Если пользователь вызывает два экземпляра прикладной программы, то вполне возможно, что обоим этим экземплярам потребуется одновременно вывести данные в файл конфигурации. В таком случае первый экземпляр прикладной программы должен заблокировать файл. И когда второй экземпляр прикладной программы обнаружит, что файл заблокирован, он может подождать до тех пор, пока файл не разблокируется, или просто пропустить процесс записи. Чтобы заблокировать файл, достаточно вызвать метод `lock()` или `tryLock()` из класса `FileChannel` следующим образом:

```
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

или таким образом:

```
FileLock lock = channel.tryLock();
```

Первый вызов блокируется до тех пор, пока блокировка не станет доступной. Возврат из второго вызова происходит немедленно со снятой блокировкой или пустым значением `null`, если блокировка недоступна. Файл остается заблокированным до тех пор, пока блокировка не снята, а канал закрыт. Поэтому для управления блокировкой файла лучше всего пользоваться оператором `try` с ресурсами, как показано ниже.

```
try (FileLock lock = channel.lock()) {
  ...
}
```

9.2. Каталоги, файлы и пути к ним

Ранее демонстрировалось, как пользоваться объектами типа `Path` для указания путей к файлам. А в последующих разделах будет показано, как манипулировать этими объектами и обращаться с каталогами и файлами.

9.2.1. Пути к файлам

Класс Path представляет путь, состоящий из последовательности имен каталогов, после которых может быть дополнительно указано имя файла. Первой составляющей пути может быть корневой каталог вроде / или C:\. Допустимые составляющие корневого каталога зависят от конкретной файловой системы. Путь, начинающийся с корневого каталога, называется *абсолютным*, а иначе — *относительным*. В качестве примера ниже демонстрируется построение абсолютного и относительного пути. При построении абсолютного пути предполагается наличие Unix-подобной файловой системы.

```
Path absolute = Paths.get("/", "home", "cay");
Path relative = Paths.get("myapp", "conf", "user.properties");
```

Статический метод Paths.get() получает в качестве параметров одну или несколько символьных строк, соединяя их разделителем пути, выбираемым в файловой системе по умолчанию (знаком / в Unix-подобной файловой системе или знаком \ в Windows). Полученный результат затем подвергается синтаксическому анализу. И если этот результат не содержит достоверный путь в данной файловой системе, то генерируется исключение типа InvalidPathException. В итоге возвращается объект типа Path.

Исходные символьные строки можно предоставить методу Paths.get() и с разделителями следующим образом:

```
Path homeDirectory = Paths.get("/home/cay");
```

НА ЗАМЕТКУ. Объект типа **Path** совсем не обязательно должен соответствовать файлу, который фактически существует. Он представляет собой всего лишь абстрактную последовательность имен. Чтобы создать файл, сначала нужно составить путь к нему, а затем вызвать метод для создания соответствующего файла, как поясняется далее, в разделе 9.2.2.

Нередко пути объединяются или "разрешаются". Так, в результате вызова метода p.resolve(q) путь возвращается в соответствии со следующими правилами.

- Если q — абсолютный путь, то и результат равен q.
- В противном случае результат равен "p затем q" согласно правилам, действующим в применяемой файловой системе.

Допустим, что в прикладной программе требуется найти файл ее конфигурации относительно начального каталога. Ниже показано, как объединить с этой целью пути.

```
Path workPath = homeDirectory.resolve("myapp/work");
    // То же, что и вызов homeDirectory.resolve(Paths.get("myapp/work"))
```

Имеется также служебный метод resolveSibling(), разрешающий заданный путь относительно родительского и возвращающий родственный путь. Так, если объект workPath представляет путь /home/cay/myapp/work, то в результате вызова

```
Path tempPath = workPath.resolveSibling("temp");
```

получается путь /home/cay/myapp/temp.

Совершенно иначе действует метод relativize(). В результате вызова p.relativize(r) получается путь q, а в результате его разрешения относительно пути p — путь r. Например, в результате следующего вызова:

```
Paths.get("/home/cay").relativize(Paths.get("/home/fred/myapp"))
```

получается относительный путь ../fred/myapp, при условии, что в применяемой файловой системе знаками .. обозначается родительский каталог.

Метод normalize() удаляет любые избыточные составляющие . и .. (или все, что файловая система может посчитать избыточным). Так, в результате нормализации пути /home/cay/../fred/./myapp получается путь /home/fred/myapp.

Метод toAbsolutePath() возвращает абсолютный путь по заданному пути. Если этот путь не является абсолютным, он разрешается относительно "пользовательского каталога", т.е. того каталога, из которого вызывается виртуальная машина JVM. Так, если запустить прикладную программу из каталога /home/cay/myapp, то в результате вызова Paths.get("config").toAbsolutePath() возвратится каталог /home/cay/myapp/config.

В интерфейсе Path имеются методы для разделения путей и последующего их соединения с другими путями. В следующем примере кода демонстрируется применение наиболее употребительных из этих методов:

```
Path p = Paths.get("/home", "cay", "myapp.properties");
Path parent = p.getParent(); // Путь /home/cay
Path file = p.getFileName(); // Последний элемент myapp.properties
                             // заданного пути
Path root = p.getRoot(); // Начальный отрезок пути / (или пустое
                         // значение null для относительного пути)
Path first = p.getName(0); // Первый элемент заданного пути
Path dir = p.subpath(1, p.getNameCount());// Весь путь, кроме
                          // первого элемента cay/myapp.properties
```

Интерфейс Path расширяет интерфейс Iterable<Path>, и поэтому составляющие имен в пути можно перебрать в расширенном цикле for следующим образом:

```
for (Path component : path) {
   ...
}
```

НА ЗАМЕТКУ. Иногда возникает потребность обращаться к унаследованным прикладным программным интерфейсам API, в которых вместо интерфейса **Path** применяется класс **File**. Если в интерфейсе **Path** имеется метод **toFile()**, то в классе **File** — метод **toPath()**.

9.2.2. Создание файлов и каталогов

Чтобы создать новый каталог, достаточно сделать следующий вызов:

```
Files.createDirectory(path);
```

Все составляющие заданного пути, кроме последней, должны непременно существовать. Чтобы создать и промежуточные каталоги, достаточно сделать такой вызов:

```
Files.createDirectories(path);
```

Пустой файл можно создать следующим образом:

```
Files.createFile(path);
```

В результате этого вызова генерируется исключение, если файл уже существует. Операции проверки существования и создания файлов являются атомарными. Если же файл не существует, то он создается, прежде чем кто-нибудь попытается сделать то же самое.

При вызове метода `Files.exists(path)` проверяется, существует ли заданный файл или каталог. Чтобы выяснить, является ли проверяемый объект каталогом или обычным файлом (т.е. он содержит данные, а не является каталогом или символической ссылкой), следует вызвать статический метод `isDirectory()` или `isRegularFile()` из класса `Files`. Для создания временного файла или каталога в заданном или характерном для системы месте имеются соответствующие служебные методы. Ниже приведен характерный пример применения этих методов.

```
Path tempFile = Files.createTempFile(dir, prefix, suffix);
Path tempFile = Files.createTempFile(prefix, suffix);
Path tempDir = Files.createTempDirectory(dir, prefix);
Path tempDir = Files.createTempDirectory(prefix);
```

В данном примере параметр `dir` обозначает объект типа `Path`, а параметры `prefix` и `suffix` — символьные строки, которые могут быть пустыми. Так, в результате вызова `Files.createTempFile(null, ".txt")` может быть возвращен путь вроде `/tmp/1234405522364837194.txt`.

9.2.3. Копирование, перемещение и удаление файлов

Чтобы скопировать файл из одного места в другое, достаточно сделать следующий вызов:

```
Files.copy(fromPath, toPath);
```

А для того чтобы переместить файл (т.е. скопировать и удалить исходный файл), достаточно сделать приведенный ниже вызов. Этим же способом можно переместить и пустой каталог.

```
Files.move(fromPath, toPath);
```

Если целевой файл существует, операция копирования или перемещения завершится неудачно. Если же требуется перезаписать существующий целевой файл, то при вызове соответствующего метода следует указать параметр `REPLACE_EXISTING`, а если требуется скопировать все атрибуты файлов — параметр `COPY_ATTRIBUTES`. Оба эти параметра можно предоставить следующим образом:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
StandardCopyOption.COPY_ATTRIBUTES);
```

Операцию перемещения файла можно указать как атомарную. В этом случае гарантируется одно из двух: операция перемещения файла завершится полностью с удачным исходом, или исходный файл будет существовать и дальше. С этой целью при вызове метода `move()` параметр `ATOMIC_MOVE` указывается следующим образом:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

В табл. 9.3 сведены параметры, доступные для операций с файлами.

Таблица 9.3. Стандартные параметры для операций с файлами

Параметр	Описание
Стандартные параметры открытия файлов: применяются месте с методами **`newBufferedWriter()`**, **`newInputStream()`**, **`newOutputStream()`**, **`write()`**	
`READ`	Открыть файл для чтения
`WRITE`	Открыть файл для записи
`APPEND`	Если файл открыт для записи, присоединить данные в конце файла
`RUNCATE_EXISTING`	Если файл открыт для записи, удалить существующее содержимое
`CREATE_NEW`	Создать новый файл, а если он уже существует, то завершить операцию с неудачным исходом
`CREATE`	Создать новый файл атомарно, а если он уже существует, то завершить операцию с неудачным исходом
`DELETE_ON_CLOSE`	Сделать все возможное, чтобы удалить файл после его закрытия
`SPARSE`	Указать файловой системе, что файл будет разреженным (т.е. неполным)
`DSYNC|SYNC`	Требует, чтобы каждое обновление содержимого файла и метаданных записывалось синхронно в запоминающем устройстве
Стандартные параметры копирования файлов: применяются вместе с методами **`copy()`** и **`move()`**	
`ATOMIC_MOVE`	Переместить файл атомарно
`COPY_ATTRIBUTES`	Копировать атрибуты файла
`REPLACE_EXISTING`	Заменить целевой файл, если он существует
Параметр перехода по ссылке: применяется вместе с упомянутыми выше методами, а также с методами **`exists()`**, **`isDirectory()`**, **`isRegularFile()`**	
`NOFOLLOW_LINKS`	Не переходить по символическим ссылкам
Параметр обхода файлов: применяется вместе с методами **`find()`**, **`walk()`**, **`walkFileTree()`**	
`FOLLOW_LINKS`	Переходить по символическим ссылкам

И наконец, чтобы удалить файл, достаточно сделать следующий вызов:

```
Files.delete(path);
```

Этот метод генерирует исключение, если файл не существует. Поэтому может потребоваться приведенный ниже вызов. Методы удаления файлов можно применять и для удаления пустого каталога.

```
boolean deleted = Files.deleteIfExists(path);
```

9.2.4. Обход элементов каталога

Статический метод `Files.list()` возвращает поток данных типа `Stream<Path>`, откуда читаются элементы каталога. Содержимое каталога читается по требованию, и благодаря этому становится возможной эффективная обработка каталогов с большим количеством элементов.

Для чтения содержимого каталога требуется закрыть системные ресурсы, и поэтому данную операцию необходимо заключить в блок оператора `try` следующим образом:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
  ...
}
```

Метод `list()` не входит в подкаталоги. Чтобы обработать все порожденные элементы каталога, следует воспользоваться методом `Files.walk()`:

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
  // Содержит все порожденные элементы, обойденные в глубину
}
```

Ниже приведен пример обхода дерева каталогов из разархивированного файла `src.zip`. Как видите, всякий раз, когда в результате обхода получается каталог, происходит вход в него, прежде чем продолжать обход родственных ему каталогов.

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
...
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
...
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
...
```

Глубину дерева каталогов, которое требуется обойти, можно ограничить, вызвав метод `Files.walk(pathToRoot, depth)`. В обоих рассматриваемых здесь вызовах метода `walk()` имеются аргументы переменной длины типа `FileVisitOption...`, но для перехода по символическим ссылкам можно предоставить только параметр `FOLLOW_LINKS`.

НА ЗАМЕТКУ. Чтобы отфильтровать пути, возвращаемые методом **`walk()`** по критерию, включающему в себя атрибуты файлов, хранящиеся в каталоге, в том числе размер, время создания или тип (файла, каталога, символической ссылки), вместо метода **`walk()`** лучше воспользоваться методом **`find()`**. Этот метод следует вызывать с предикатной функцией, принимающей в качестве параметров путь и объект типа **`BasicFileAttributes`**. Единственное преимущество такого подхода состоит в его эффективности. Ведь чтение каталога происходит в любом случае, и поэтому атрибуты становятся сразу же доступными.

В следующем фрагменте кода метод `Files.walk()` применяется для копирования одного каталога в другой:

```
Files.walk(source).forEach(p -> {
  try {
    Path q = target.resolve(source.relativize(p));
    if (Files.isDirectory(p))
      Files.createDirectory(q);
    else
      Files.copy(p, q);
  } catch (IOException ex) {
    throw new UncheckedIOException(ex);
  }
});
```

К сожалению, методом `Files.walk()` не так-то просто воспользоваться для удаления дерева каталогов, поскольку для этого нужно обойти все порожденные элементы, прежде чем удалить родительский. В таком случае лучше воспользоваться методом `walkFileTree()`, которому требуется экземпляр класса, реализующего интерфейс `FileVisitor`. Ниже перечислены случаи, когда извещается метод обхода файлов.

1. Перед обработкой каталога:

   ```
   FileVisitResult preVisitDirectory(T dir, IOException ex)
   ```

2. Когда встречается файл или каталог:

   ```
   FileVisitResult visitFile(T path, BasicFileAttributes attrs)
   ```

3. Когда в методе `visitFile()` возникает исключение:

   ```
   FileVisitResult visitFileFailed(T path, IOException ex)
   ```

4. После обработки каталога:

   ```
   FileVisitResult postVisitDirectory(T dir, IOException ex)
   ```

В каждом из перечисленных случаев извещающий метод возвращает один из следующих результатов.

- Продолжить обход со следующего файла:

  `FileVisitResult.CONTINUE`.

- Продолжить обход, но не обращаясь к элементам данного каталога:

  `FileVisitResult.SKIP_SUBTREE`

- Прекратить обход:

  `FileVisitResult.TERMINATE`.

Обход прекращается и в том случае, если любой из перечисленных выше методов генерирует исключение, причем это исключение генерируется в теле метода `walkFileTree()`. Интерфейс `FileVisitor` реализуется в классе `SimpleFileVisitor`, где обход дерева каталогов продолжается, и на каждой стадии повторно генерируются любые исключения.

В приведенном ниже примере показано, каким образом удаляется дерево каталогов.

```
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
  public FileVisitResult visitFile(Path file,
        BasicFileAttributes attrs) throws IOException {
    Files.delete(file);
    return FileVisitResult.CONTINUE;
}
public FileVisitResult postVisitDirectory(Path dir,
       IOException ex) throws IOException {
    if (ex != null) throw ex;
    Files.delete(dir);
    return FileVisitResult.CONTINUE;
  }
});
```

9.2.5. Системы файлов формата ZIP

Средствами класса `Paths` осуществляется поиск путей в файловой системе, применяемой по умолчанию, т.е. среди пользовательских файлов на локальном диске. Но ведь могут применяться и другие файловые системы. К числу наиболее полезных относится система архивных файлов формата ZIP. Так, если `zipname` — это имя архивного ZIP-файла, то в результате следующего вызова:

```
FileSystem zipfs = FileSystems.newFileSystem(Paths.get(zipname), null);
```

устанавливается файловая система, содержащая все файлы в архиве формата ZIP. Если известно имя этого архива, то скопировать файл из него не составит большого труда, как показано ниже. В данном случае метод `zipfs.getPath()` действует аналогично методу `Paths.get()` для произвольной файловой системы.

```
Files.copy(zipfs.getPath(sourceName), targetPath);
```

Чтобы перечислить все файлы в архиве формата ZIP, придется обойти дерево файлов следующим образом:

```
Files.walk(zipfs.getPath("/")).forEach(p -> {
  Обработать узел дерева p
});
```

А для того чтобы создать новый ZIP-файл, придется потрудиться чуть больше. Ниже показано, как это осуществляется на практике.

```
Path zipPath = Paths.get("myfile.zip");
URI uri = new URI("jar", zipPath.toUri().toString(), null);
  // построить URI по образцу jar:file://myfile.zip
try (FileSystem zipfs = FileSystems.newFileSystem(uri,
        Collections.singletonMap("create", "true"))) {
  // скопировать файлы в систему файлов формата ZIP, чтобы ввести их
  Files.copy(sourcePath, zipfs.getPath("/").resolve(targetPath));
}
```

НА ЗАМЕТКУ. Имеется еще один устаревший прикладной программный интерфейс API для обращения с архивами формата ZIP. В нем имеются классы **`ZipInputStream`** и **`ZipOutputStream`**, но пользоваться ими не так просто, как описано выше.

9.5. Подключения по заданному URL

Чтобы выполнить чтение данных из веб-ресурса по указанному URL, следует вызвать метод `getInputStream()` для объекта типа `URL`. А для получения дополнительных сведений о веб-ресурсе или записи в него данных служит класс `URLConnection`. Ниже описывается соответствующий порядок действий.

1. Получить объект типа `URLConnection`:

   ```
   URLConnection connection = url.openConnection();
   ```

   Для URL формата HTTP возвращается объект, который, по существу, является экземпляром класса `HttpURLConnection`.

2. Задать, если требуется, свойства запроса:

   ```
   connection.setRequestProperty("Accept-Charset", "UTF-8, ISO-8859-1");
   ```

   Если ключу соответствует несколько значений, разделить их запятыми.

3. Сделать следующий вызов, чтобы отправить данные на сервер:

   ```
   connection.setDoOutput(true);
   try (OutputStream out = connection.getOutputStream()) {
     Вывести данные в поток вывода out
   }
   ```

4. Если требуется прочитать заголовки из ответа, а метод `getOutputStream()` еще не вызывался, то сделать следующий вызов:

   ```
   connection.connect();
   ```

   Затем запросить данные из заголовков следующим образом:

   ```
   Map<String, List<String>> headers = connection.getHeaderFields();
   ```

   По каждому ключу получается список значений, поскольку одному и тому же ключу может соответствовать несколько полей в заголовках.

5. Прочитать ответ:

```
try (InputStream in = connection.getInputStream()) {
  Ввести данные из потока ввода in
}
```

Подключение по заданному URL чаще всего применяется для отправки данных заполненной формы. В классе `URLConnection` автоматически задается тип содержимого `application/x-www-form-urlencoded` для вывода данных по заданному URL формата HTTP, но пары "имя–значение" придется закодировать, как показано ниже.

```
URL url = ...;
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
try (Writer out = new OutputStreamWriter(
     connection.getOutputStream(), StandardCharsets.UTF_8)) {
  Map<String, String> postData = ...;
  boolean first = true;
  for (Map.Entry<String, String> entry : postData.entrySet()) {
    if (first) first = false;
    else out.write("&");
    out.write(URLEncoder.encode(entry.getKey(), "UTF-8"));
    out.write("=");
    out.write(URLEncoder.encode(entry.getValue(), "UTF-8"));
  }
}
try (InputStream in = connection.getInputStream()) {
  ...
}
```

9.4. Регулярные выражения

Регулярные выражения обозначают шаблоны символьных строк. Они применяются всякий раз, когда требуется найти символьные строки, совпадающие с определенным шаблоном. Допустим, что требуется найти гиперссылки в HTML-файле и, в частности, символьные строки, совпадающие с шаблоном `<a href="...">`. Но ведь в этих строках могут быть лишние пробелы, а URL может быть заключен в одиночные кавычки. Регулярные выражения предоставляют точный синтаксис для обозначения тех последовательностей символов, которые допускаются при совпадении. В последующих разделах будет представлен синтаксис регулярных выражений, принятый в прикладном программном интерфейсе Java API, а также рассмотрены особенности применения регулярных выражений на практике.

9.4.1. Синтаксис регулярных выражений

Символ в регулярном выражении имеет свое первоначальное обозначение, если только он не относится к одному из следующих зарезервированных символов:

```
. * + ? { | ( ) [ \ ^ $
```

Например, регулярное выражение **Java** обозначает совпадение только с символьной строкой `"Java"`, а знак `.` — совпадение с любым символом. Так, регулярное выражение **.a.a** совпадает с символьными строками `"Java"` и `"data"`. Знак `*` обозначает,

что предыдущие конструкции могут *не* повторяться или же повторяться многократно, а знак — повторение 1 и больше раз. Суффикс ? указывает на то, что конструкция является дополнительной и может *не* повторяться или же повторяться один раз. Например, регулярное выражение `be+s?` совпадает с символьными строками "be", "bee" и "bees". В фигурных скобках { } можно указать и другие виды многократного совпадения, как поясняется в табл. 9.4.

Знак | обозначает альтернативный выбор. Так, регулярное выражение `.(oo|ee)f` совпадает с символьной строкой "beef" или "woof". Обратите внимание на круглые скобки. Без них регулярное выражение `.oo|eef` обозначало бы альтернативный выбор между совпадением с шаблоном `.oo` или `eef`. Круглые скобки употребляются также для группирования, как поясняется далее, в разделе 9.4.3.

Класс символов представляет собой набор альтернативных символов, заключаемых в квадратные скобки, например, `[Jj]`, `[0-9]`, `[A-Za-z]` или `[^0-9]`. В самом классе символов знак - обозначает диапазон символов, т.е. все символы, значения которых в Юникоде оказываются в указанных пределах. Но если знак - указан в классе символов первым или последним, то он имеет свое первоначальное значение. А знак ^, указанный первым в классе символов, обозначает дополнение, т.е. все символы, кроме указанных.

Имеется немало *предопределенных классов символов*, например, `\d` (для цифр) или `\p{Sc}` для знаков денежных единиц в Юникоде, как поясняется в табл. 9.4 и 9.5. А знаки ^ и $ обозначают совпадение с началом и концом ввода данных соответственно.

Если знаки `.`, `*`, `+`, `?`, `{`, `|`, `(`, `)`, `[`, `\`, `^`, `$` требуется указать буквально, их следует предварить знаком обратной косой черты. Но в самом классе символов такое экранирование требуется только для знаков `[` и `\` с учетом расположения знаков `]` - `^`. Так, класс символов `[]^-]` содержит все эти знаки.

Таблица 9.4. Синтаксис регулярных выражений

Выражение	Описание	Пример
	Символы	
c, кроме знаков * + ? { | ( ) [\ ^ $	Символ *c*,	*J*
.	Любой символ, кроме знаков окончания строки, или любой символ, если установлен признак `DOTALL`	
`\x{`*p*`}`	Кодовая точка в Юникоде с шестнадцатеричным значением *p*	`\x{1D546}`
`\u`*hhhh*, `\x`*hh*, `\0`*o*, `\0`*oo*, `\0`*ooo*	Кодовая точка в кодировке UTF-16 с указанным шестнадцатеричным или восьмеричным значением	`\uFEFF`
`\a`, `\e`, `\f`, `\n`, `\r`, `\t`	Предупреждение (`\x{7}`), экранирование (`\x{1B}`), перевод страницы (`\x{B}`), новая строка (`\x{A}`), перевод каретки (`\x{D}`), табуляция (`\x{9}`)	`\n`
`\c`*c*, где *c* — любой из символов `[A-Z]` или знаков `@ [ \ ] ^ _ ?`	Управляющий символ, соответствующий указанному символу *c*	`\cH` — возврат на одну позицию назад (`\x{8}`)

Продолжение табл. 9.4

Выражение	Описание	Пример
`\c`, где *c* — любой символ, кроме `[A-Za-z0-9]`	Символ *c*	`\\`
`\Q . . . \E`	Все, что находится между началом и концом цитирования	`\Q(...)\E` обозначает совпадение с символьной строкой `"(...)"`
	Классы символов	
`[`C_1C_2 `. . . ]`, где C_i — символы, диапазоны *c*-*d* или классы символов	Любой из символов `[0-9+-]`, обозначаемых как C_1, C_2, ...	`[0-9+-]`
`[^ . . . ]`	Дополнение класса символов	`[^\d\s]`
`[ . . . && . . . ]`	Пересечение классов символов	`[\p{L}&&[^A-Za-z]]`
`\p{ . . . }`, `\P{ . . . }`	Предопределенный класс символов (см. табл. 9.5); служит дополнением	`\p{L}` или `\pL` обозначает совпадение с указанной буквой в Юникоде; фигурные скобки можно опустить
`\d`, `\D`	Цифры `[0-9]` или `\p{Digit}`, если установлен признак `UNICODE_CHARACTER_CLASS`; служит дополнением	`\d+` обозначает совпадение с последовательностью цифр
`\w`, `\W`	Словесные символы (`[a-zA-Z0-9_]` или словесные символы в Юникоде, если установлен признак `UNICODE_CHARACTER_CLASS`); служит дополнением	
`\s`, `\S`	Пробелы (`[ \n\r\t\f\x{B}]` или `\p{IsWhite_Space}`, если установлен признак `UNICODE_CHARACTER_CLASS`); служит дополнением	`\s*`, `\s*` обозначает запятую с дополнительными пробелами вокруг нее
`\h`, `\v`, `\H`, `\V`	Пробелы по горизонтали или по вертикали или же их дополнения	
	Последовательности символов и их альтернативы	
XY	Любая строка из последовательности символов *X*, после которой следует любая строка из последовательности символов *Y*	`[1-9][0-9]*` обозначает положительное число без начального нуля
X|*Y*	Любая строка из последовательности символов *X* или *Y*	`http|ftp`
	Группирование	
`(`*X*`)`	Фиксирует совпадение с *X*	`'([^']*)'` фиксирует цитируемый текст
`\`*n*	*n*-я группа	`(['"]).*\1` обозначает совпадение со строкой `'Fred'` или `"Fred"`, но не `"Fred'`

Окончание табл. 9.4

Выражение	Описание	Пример
`(?<`*имя*`>`*X*`)`	Фиксирует совпадение *X* с указанным именем	`'(?<id>[A-Za-z0-9]+)'` фиксирует совпадение с именем `id`
`\k<`*имя*`>`	Группа с заданным именем	`\k<id>` обозначает совпадение с группой под именем `id`
`(?:`*X*`)`	Круглые скобки употребляются без фиксации *X*	Совпадение `\1` в группе `(?:http|ftp)://(.*)` после знаков `://`
`(?`$f_1f_2 \ldots$`:`*X*`)`, `(?`$f_1 \ldots -f_k \ldots$`:`*X*`)`, где f_i относится к `[dimsuUx]`	Совпадение, но не фиксация, где *X* указывается с установленными или сброшенными (после знака –) признаками	`(?i:jpe?g)` обозначает совпадение без учета регистра букв
Другое `(?...)`	См. документацию на класс `Pattern`	
	Кванторы	
X`?`	Дополнительно *X*	`\+?` обозначает дополнительный знак "плюс"
X`*`, *X*`+`	0 или больше повторений *X*, 1 или больше повторений *X*	`[1-9][0-9]` обозначает целое число, равное или большее 10
`X{`*n*`}`, `X{`*n*`,}`, `X{`*m*`,`*n*`}`	*n* повторений *X*, не меньше *n* повторений *X*, от *m* до *n* повторений *X*	`.*(<.+?>).*` фиксирует кратчайшую последовательность символов в угловых скобках
Q`+`, где Q — кванторное выражение	Притяжательный квантор, принимающий самое длинное совпадение без возврата	`'[^']*+'` обозначает совпадение со строками в одиночных кавычках, но не со строками без одиночных кавычек
Совпадение в заданных пределах		
`^ $`	Начало и конец ввода данных (или начало и конец строки в многострочном режиме)	`^Java$` обозначает совпадение с вводимыми данными или строкой `"Java"`
`\A \Z \z`	Начало ввода, конец ввода, абсолютный конец ввода (не изменяется в многострочном режиме)	
`\b \B`	Граница слова, граница неслова	`\bJava\b` обозначает совпадение со словом `Java`
`\R`	Разрыв строки в Юникоде	
`\G`	Конец предыдущего совпадения	

Таблица 9.5. Предопределенные классы символов `\p{...}`

Наименование	Описание
posixClass	*posixClass* обозначает один из классов **Lower**, **Upper**, **Alpha**, **Digit**, **Alnum**, **Punct**, **Graph**, **Print**, **Cntrl**, **XDigit**, **Space**, **Blank**, **ASCII**, интерпретируемых как класс по стандарту POSIX или Unicode, в зависимости от состояния признака **UNICODE_CHARACTER_CLASS**
Is*Script*, **sc=***Script*, **script=***Script*	Сценарий, принимаемый методом **Character.UnicodeScript.forName()**
In*Block*, **blk=***Block*, **block=***Block*	Блок кода, принимаемый методом **Character.UnicodeBlock.forName()**
Category, **In***Category*, **gc=***Category*, **general_category=***Category*	Одно- или двухбуквенное наименование общей категории в Юникоде
Is*Property*	*Property* обозначает одно из свойств **Alphabetic**, **Ideographic**, **Letter**, **Lowercase**, **Uppercase**, **Titlecase**, **Punctuation**, **Control**, **White_Space**, **Digit**, **Hex_Digit**, **Join_Control**, **Noncharacter_Code_Point**, **Assigned**
java*Method*	Вызывает метод **Character.is***Method***()**, который должен быть рекомендован к употреблению

9.4.2. Обнаружение одного или всех совпадений

Как правило, регулярным выражением можно воспользоваться следующими двумя способами: с целью выявить соответствие символьной строки заданному регулярному выражению или все совпадения с регулярными выражениями в символьной строке. В первом случае просто употребляется статический метод `matches()`, как показано ниже.

```
String regex = "[+-]?\\d+";
CharSequence input = ...;
if (Pattern.matches(regex, input)) {
   ...
}
```

Если одно и то же регулярное выражение требуется употребить неоднократно, то его эффективнее сначала скомпилировать, а затем создать объект типа `Matcher` для каждого ввода данных, как показано в следующем примере кода:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) ...
```

Если совпадение оказывается успешным, то местоположение совпавших групп можно извлечь, как поясняется в следующем разделе. Если же требуется выявить совпадение элементов в коллекции или потоке данных, то шаблон следует превратить в предикат, как показано ниже. А результат содержит все символьные строки, совпадающие с заданным регулярным выражением.

```
Stream<String> strings = ...;
Stream<String> result = strings.filter(pattern.asPredicate());
```

А теперь рассмотрим другой случай: поиск всех совпадений с регулярным выражением во вводимой символьной строке. Для этой цели применяется следующий цикл:

```
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
  String match = matcher.group();
  ...
}
```

Подобным образом можно обработать каждое совпадение по очереди. В следующем разделе будет показано, каким образом все совпадения обрабатываются вместе.

9.4.3. Группы

Группы обычно применяются для извлечения составляющих совпадения. Допустим, что имеется позиция в счете-фактуре с наименованием товара, количеством и ценой за штуку:

```
Blackwell Toaster USD29.95
```

Следующее регулярное выражение содержит группы для каждой составляющей совпадения:

```
(\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

После совпадения можно извлечь *n*-ю группу из сопоставителя с шаблоном следующим образом:

```
String contents = matcher.group(n);
```

Группы упорядочиваются по открывающей круглой скобке, начиная с 1. (Нулевая группа обозначает все вводимые данные.) В следующем примере показано, каким образом разделяются вводимые данные:

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
  item = matcher.group(1);
  currency = matcher.group(3);
  price = matcher.group(4);
}
```

Группа 2 нас не интересует, поскольку она возникла из круглых скобок, потребовавшихся только для повторения. Для большей ясности можно воспользоваться нефиксирующей группой следующим образом:

```
(\p{Alnum}+(?:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

А еще лучше произвести фиксацию по имени таким образом:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<currency>[A-Z]{3})(?<price>[0-9.]*)
```

Затем отдельные элементы можно извлечь по имени, как показано ниже.

```
item = matcher.group("item");
```

НА ЗАМЕТКУ. Если в повторении присутствует группа вроде `(\s+\p{Alnum})*` из приведенного выше примера, то получить все эти совпадения невозможно. Метод `group()` возвращает только последнее совпадение, от чего редко бывает польза. Поэтому выражение нужно полностью зафиксировать с помощью другой группы.

9.4.4. Удаление и замена совпадений

Иногда вводимые данные требуется расчленить совпадающими разделителями, сохранив все остальное. Эту задачу позволяет решить метод `Pattern.split()`. Так, массив символьных строк с удаленными разделителями можно получить следующим образом:

```
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input);
  // Строка "1, 2, 3" превращается в строку ["1", "2", "3"]
```

Если же имеется много лексем, их можно извлечь по требованию таким образом:

```
Stream<String> tokens = commas.splitAsStream(input);
```

Если предварительная компиляция шаблона или извлечение по требованию не имеет значения, то можно просто воспользоваться методом `String.split()` следующим образом:

```
String[] tokens = input.split("\\s*,\\s*");
```

Если же требуется заменить все совпадения с символьной строкой, то для сопоставителя с шаблоном достаточно вызвать метод `replaceAll()`, как показано ниже.

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
  // Нормализует запятые
```

А если не имеет значения предварительная компиляция шаблона, то для замены всех совпадений с символьной строкой можно вызвать метод `replaceAll()` из класса `String` следующим образом:

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

Заменяющая строка может содержать номера групп `$n` или их имена `${имя}`. Они заменяются содержимым соответствующей зафиксированной группы, как показано ниже. Знак `\` служит для экранирования знаков `$` и `\` в заменяющей строке.

```
String result = "3:45".replaceAll(
  "(\\d{1,2}):(?<minutes>\\d{2})",
  "$1 hours and ${minutes} minutes");
  // Задает следующий результат: "3 hours and 45 minutes"
```

9.4.5. Признаки

Имеется ряд *признаков* (или так называемых *флагов*), с помощью которых можно изменить поведение регулярных выражений. Признаки можно указать при компиляции шаблона следующим образом:

```
Pattern pattern = Pattern.compile(regex,
    Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLASS);
```

С другой стороны, их можно указать в самом шаблоне таким образом:

```
String regex = "(?iU:expression)";
```

Ниже перечислены доступные признаки.

- **Pattern.CASE_INSENSITIVE** или **i**. Задает совпадение с символами без учета регистра букв. По умолчанию этот признак принимает во внимание только символы в коде US ASCII.
- **Pattern.UNICODE_CASE** или **u**. В сочетании с признаком **CASE_INSENSITIVE** этот признак задает совпадение с символами букв в Юникоде.
- **Pattern.UNICODE_CHARACTER_CLASS** или **U**. Задает выбор классов символов в Юникоде, а не по стандарту POSIX. Признак **UNICODE_CASE** подразумевается.
- **Pattern.MULTILINE** или **m**. Задает совпадение знаков **^** и **$** с началом и концом строки, а не со всеми вводимыми данными.
- **Pattern.UNIX_LINES** или **d**. Задает последовательность символов **'\n'** в качестве единственного признака окончания строки при совпадение знаков **^** и **$** с началом и концом строк в многострочном режиме.
- **Pattern.DOTALL** или **s**. Задает совпадение знака **.** со всеми символами, включая и знаки конца строки.
- **Pattern.COMMENTS** или **x**. Задает игнорирование пробела и комментариев (от знака **#** и до конца строки).
- **Pattern.LITERAL**. Задает буквальное восприятие шаблона и точное совпадение с ним, кроме тех случаев, когда требуется учитывать регистр.
- **Pattern.CANON_EQ**. Задает режим, в котором во внимание принимается каноническая эквивалентность символов в Юникоде. Например, последовательность символов **u** и ¨ (диерезис) обозначает совпадение с символом **ü**.

Два последних признака могут быть указаны в самом регулярном выражении.

9.5. Сериализация

В последующих разделах рассматривается сериализация — механизм преобразования объектов в последовательность байтов, которые могут быть переданы куда-нибудь или сохранены на диске. В этих разделах рассматривается и обратный механизм восстановления объектов из последовательности байтов.

Сериализация является важным средством для организации распределенной обработки данных, где объекты передаются из одной виртуальной машины на другую. Она применяется также для восстановления после отказа и выравнивания нагрузки, когда сериализированные объекты могут быть перемещены на другой сервер. Если приходится иметь дело с программным обеспечением серверов, то для классов нередко требуется активизировать сериализацию. В последующих разделах поясняется, как это делается.

9.5.1. Интерфейс Serializable

Для сериализации объекта (т.е. его преобразования в последовательность байтов) он должен быть экземпляром класса, реализующего интерфейс `Serializable`. Это маркерный интерфейс без методов, аналогичный интерфейсу `Cloneable`, упоминавшемуся в главе 4.

Например, чтобы сделать объект класса `Employee` сериализируемым, этот класс должен быть объявлен следующим образом:

```
public class Employee implements Serializable {
  private String name;
  private double salary;
  ...
}
```

Интерфейс `Serializable` можно благополучно реализовать, если все переменные экземпляра относятся к примитивному или перечислимому типу либо ссылаются на сериализируемые объекты. Многие классы из стандартной библиотеки Java допускают сериализацию. Массивы и классы коллекций, представленные в главе 7, также допускают сериализацию, если сериализуемы их элементы. Чаще всего все объекты, доступные из сериализируемого объекта, также должны быть сериализируемыми.

Если вернуться к примеру класса `Employee`, то сериализация его объектов, как, впрочем, и объектов большинства классов, не вызывает особых трудностей. В последующих разделах будет показано, что нужно сделать, если потребуется небольшая дополнительная помощь в сериализации.

Для сериализации объектов требуется поток вывода типа `ObjectOutputStream`, который организуется с помощью другого потока вывода типа `OutputStream`, получающего конкретные байты, как показано ниже.

```
ObjectOutputStream out = new ObjectOutputStream(
    Files.newOutputStream(path));
```

Затем вызывается метод `writeObject()`, как показано в следующем примере кода:

```
Employee peter = new Employee("Peter", 90000);
Employee paul = new Manager("Paul", 180000);
out.writeObject(peter);
out.writeObject(paul);
```

Чтобы ввести объекты обратно, нужно организовать поток ввода типа `ObjectInputStream` следующим образом:

```
ObjectInputStream in = new ObjectInputStream(
Files.newInputStream(path));
```

Объекты извлекаются из потока ввода в том порядке, в каком они выводились. Для этой цели служит метод `readObject()`:

```
Employee e1 = (Employee) in.readObject();
Employee e2 = (Employee) in.readObject();
```

При выводе объекта сохраняется имя его класса, а также имена и значения всех переменных экземпляра. Если значение переменной экземпляра относится к примитивному типу, оно сохраняется в двоичном формате данных. А если это объект, то он выводится с помощью метода `writeObject()`, как было показано выше.

Когда объект читается из потока ввода, все происходит в обратном порядке. Сначала читается имя класса вместе с именами и значениями переменных экземпляра, а затем восстанавливается сам объект, но с одной оговоркой. Допустим, что имеются две ссылки на один и тот же объект, например, ссылки работника на его начальников. При чтении обоих приведенных ниже объектов обратно из потока ввода они должны иметь ссылку *на одного и того же* начальника, а не две ссылки на одинаковые, но отдельные объекты.

```
Employee peter = new Employee("Fred", 90000);
Employee paul = new Manager("Barney", 105000);
Manager mary = new Manager("Mary", 180000);
peter.setBoss(mary);
paul.setBoss(mary);
out.writeObject(peter);
out.writeObject(paul);
```

Чтобы добиться этого, каждый объект получает *серийный номер* при его сохранении. При передаче ссылки на объект методу `writeObject()` в потоке вывода типа `ObjectOutputStream` проверяется, была ли раньше записана ссылка на объект. В этом случае просто записывается серийный номер, а содержимое объекта не дублируется.

Аналогично в потоке вывода типа `ObjectInputStream` запоминаются все встречающиеся в нем объекты. А при обратном чтении ссылки на повторяющийся объект просто предоставляется ссылка на прочитанный раньше объект.

9.5.2. Переходные переменные экземпляра

Некоторые переменные экземпляра не подлежат сериализации. Например, сериализация соединений с базой данных теряет всякий смысл при восстановлении объекта. А если в объекте хранится кеш значений, то вместо хранения значений в кеше лучше пересчитать их снова.

Чтобы предотвратить сериализацию переменной экземпляра, достаточно пометить ее модификатором доступа `transient`. Переменные экземпляра помечаются как переходные и в том случае, если они относятся к несериализируемым классам. Переходные поля всегда пропускаются, когда объекты сериализируются.

9.5.3. Методы readObject() и writeObject()

В редких случаях механизм сериализации требует специальной настройки. Стандартные операции ввода-вывода можно дополнить любым требующимся действием, определив в сериализируемом классе методы со следующими сигнатурами:

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
private void writeObject(ObjectOutputStream out)
    throws IOException
```

В таком случае заголовки объектов выводятся как обычно, но переменные экземпляра (или поля) не сериализируются автоматически. Вместо этого вызываются упомянутые выше методы.

Рассмотрим типичный пример. Класс Point2D из библиотеки JavaFX не подлежит сериализации. Допустим, что требуется сериализация объектов класса LabeledPoint, хранящих объекты типа String и Point2D. Прежде всего нужно пометить поле типа Point2D как transient во избежание исключения типа NotSerializableException:

```
public class LabeledPoint implements Serializable {
  private String label;
  private transient Point2D point;
  ...
}
```

В методе writeObject() нужно сначала вывести непереходную переменную label, вызвав метод defaultWriteObject(). Это специальный метод из класса ObjectOutputStream, который вызывается только из метода writeObject() сериализируемого класса. Затем следует вывести координаты точки, используя метод writeDouble() из интерфейса DataOutput, как показано ниже.

```
private void writeObject(ObjectOutputStream out)
        throws IOException {
  out.defaultWriteObject();
  out.writeDouble(point.getX());
  out.writeDouble(point.getY());
}
```

В методе readObject() все происходит в обратном порядке:

```
private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
  in.defaultReadObject();
  double x = in.readDouble();
  double y = in.readDouble();
  point = new Point2D(x, y);
}
```

В методах readObject() и writeObject() нужно организовать ввод-вывод переменных экземпляра только их собственного класса, но не данных из их суперкласса.

НА ЗАМЕТКУ. В классе можно определить свой формат сериализации, реализовав интерфейс **Externalizable** и предоставив следующие методы:

```
public void readExternal(ObjectInput in)
public void writeExternal(ObjectOutput out)
```

При чтении сериализируемого объекта в потоке ввода сначала создается объект с помощью конструктора без аргументов, а затем вызывается метод **`readExternal()`**. И хотя такой прием позволяет повысить производительность, он применяется крайне редко.

9.5.4. Методы `readResolve()` и `writeReplace()`

Само собой разумеется, что объекты могут быть построены только с помощью конструктора. Но десериализируемый объект *нельзя* построить подобным образом. Его переменные экземпляра просто восстанавливаются из потока ввода объектов.

Это может вызвать трудности, если в конструкторе соблюдается некоторое условие. Например, одноэлементный объект может быть реализован таким образом, чтобы вызывать конструктор только один раз. До появления в Java перечислений перечислимые типы имитировались классами с закрытым конструктором, вызывавшимся один раз для каждого экземпляра. В качестве другого примера записи в базе данных могли быть построены таким образом, чтобы всегда поступать из пула управляемых экземпляров.

Теперь подобные ситуации возникают крайне редко. Сериализация объектов перечислимого типа происходит автоматически, и для одноэлементных объектов не нужно реализовывать свой механизм. Так, если требуется одноэлементный объект, достаточно создать один экземпляр перечислимого типа, который условно называется `INSTANCE`, как показано ниже.

```
public enum PersonDatabase {
  INSTANCE;

  public Person findById(int id) { ... }
  ...
}
```

А теперь допустим, что возникла редкая ситуация, когда требуется контролировать идентичность каждого десериализируемого экземпляра. В качестве примера рассмотрим восстановление экземпляров класса `Person` из базы данных в процессе десериализации. В таком случае сериализации подлежит не сам объект, но некоторый его заместитель, способный найти или построить объект. С этой целью следует предоставить метод `writeReplace()`, возвращающий объект-заместитель, как показано ниже.

```
public class Person implements Serializable {
  private int id;
  // Другие переменные экземпляра
  ...
  private Object writeReplace() {
    return new PersonProxy(id);
  }
}
```

Когда происходит сериализация объекта типа Person, ни одна из его переменных экземпляра не сохраняется. Вместо этого вызывается метод writeReplace() и сериализируется *возвращаемое им значение*, которое затем направляется в поток вывода. А в классе-заместителе нужно реализовать метод readResolve(), возвращающий объект типа Person, как показано ниже. Если метод readObject() обнаруживает объект типа PersonProxy в потоке ввода типа ObjectInputStream, он производит десериализацию объекта-заместителя, вызывает свой метод readResolve() и возвращает результат.

```
public class PersonProxy implements Serializable {
  private int id;

  public PersonProxy(int id) {
    this.id = id;
  }

  public Object readResolve() {
    return PersonDatabase.INSTANCE.findById(id);
  }
}
```

9.5.5. Контроль версий

Сериализация предназначалась для передачи объектов из одной виртуальной машины на другую или для кратковременного сохранения их состояния. Если же сериализация применяется для долговременного хранения или в любой ситуации, когда создаваемые классы могут попеременно производить то сериализацию, то десериализацию, это обстоятельство придется учитывать при дальнейшем развитии классов. В частности, могут ли старые данные быть прочитаны в версии 2, а пользователи версии 1 — читать файлы, созданные в новой версии?

Механизм сериализации поддерживает простую схему контроля версий. Когда происходит сериализация объекта, имя класса и его идентификатор serialVersionUID направляются в поток вывода. Этот однозначный идентификатор присваивается разработчиком класса, определяющим следующую переменную экземпляра:

```
private static final long serialVersionUID = 1L; // Версия 1
```

При дальнейшем развитии класса по несовместимому пути его разработчику придется изменить идентификатор UID. Всякий раз, когда дересериализируемый объект имеет несовпадающий идентификатор UID, метод readObject() генерирует исключение типа InvalidClassException.

Если идентификатор serialVersionUID совпадает, то десериализация продолжается, даже если реализация изменилась. В каждой непереходной переменной экземпляра вводимого объекта должно быть установлено сериализируемое состояние, при условии, что имя и тип совпадают. А во всех остальных переменных экземпляра устанавливаются значения по умолчанию: пустое значение null для ссылок на объекты, нулевое значение для чисел и логическое значение false для значений типа boolean. Все, что относится к сериализируемому состоянию и отсутствует во вводимом объекте, просто игнорируется.

Насколько этот процесс безопасный? Об этом известно только разработчику класса. Если этот процесс все же безопасный, то разработчик должен присвоить новой версии класса такой же самый идентификатор `serialVersionUID`, как и в прежней версии.

Если не присвоить идентификатор `serialVersionUID`, он автоматически генерируется хешированием канонического описания переменных экземпляра, методов и супертипов. Полученный в итоге хеш-код можно посмотреть с помощью утилиты **`serialver`**. Так, по команде

```
serialver ch09.sec05.Employee
```

отображается такой результат:

```
private static final long serialVersionUID = -4932578720821218323L;
```

Когда реализация класса изменяется, то существует большая вероятность того, что изменится и хеш-код. Если же требуется прочитать экземпляры прежней версии класса и есть уверенность, что это можно сделать надежно, то утилиту **`serialver`** следует выполнить с прежней версией класса и добавить полученный результат к новой его версии.

НА ЗАМЕТКУ. Если требуется реализовать более сложную схему контроля версий, следует переопределить метод **`readObject()`** и вызвать метод **`readFields()`** вместо метода **`defaultReadObject()`**. В итоге будет получено описание всех полей, обнаруженных в потоке ввода, после чего их можно обработать как угодно.

Упражнения

1. Напишите служебный метод для копирования всего содержимого из потока ввода `InputStream` в поток вывода `OutputStream`, не пользуясь временными файлами. Предоставьте другое решение без цикла, используя методы из класса `Files` и временный файл.
2. Напишите программу для чтения текстового файла и создания файла с таким же именем, но с расширением **`.toc`**. Этот файл должен содержать список всех слов из входного файла, а также список номеров строк, в которых встречается каждое слово. Имейте в виду, что содержимое входного файла представлено в кодировке UTF-8.
3. Напишите программу для чтения текстового файла, содержащего слова в основном на английском языке, и определения типа кодировки: ASCII, ISO 8859-1, UTF-8 или UTF-16. Если это кодировка UTF-16, то программа должна определить порядок следования байтов.
4. Пользоваться классом `Scanner` удобно, но он действует чуть медленнее, чем класс `BufferedReader`. Организуйте построчное чтение длинного файла, подсчитывая количество вводимых строк, во-первых, с помощью класса `Scanner` и методов `hasNextLine()` и `nextLine()`; во-вторых, с помощью

класса BufferedReader и метода readLine(); а в-третьих, с помощью класса BufferedReader и метода lines(). Каким из способов читать из файла быстрее и удобнее всего?

5. Если кодировщик типа Charset лишь частично охватывает Юникод и не в состоянии закодировать символ, то он заменяется символом по умолчанию — как правило, хотя и не всегда, знаком ?. Найдите замены всех доступных наборов символов, поддерживающих кодировку. Воспользуйтесь методом newEncoder(), чтобы получить кодировщик, и вызовите его метод replacement(), чтобы получить соответствующую замену. Для каждого однозначного результата выведите отчет о канонических именах, используемых для замены наборов символов.

6. Формат BMP для файлов несжатых изображений хорошо документирован и прост. Используя произвольный доступ, напишите программу, отражающую положение каждого ряда пикселей, не прибегая к записи в новый файл.

7. Просмотрите документацию на класс MessageDigest и напишите программу, вычисляющую свертку файла по алгоритму SHA-1. Снабдите блоками байтов объект типа MessageDigest с помощью метода update(), а затем отобразите результат вызова метода digest(). Убедитесь в том, что написанная вами программа выдает такой же результат, как и утилита **sha1sum**.

8. Напишите служебный метод для создания архивного ZIP-файла, содержащего все файлы из выбранного каталога и его подкаталогов.

9. Используя класс URLConnection, введите данные с веб-страницы, защищенной паролем с элементарной аутентификацией. Соедините вместе имя пользователя, двоеточие, пароль и определите кодировку Base64 полученного результата следующим образом:

```
String input = username + ":" + password;
String encoding = Base64.getEncoder().encodeToString(
    input.getBytes(StandardCharsets.UTF_8));
```

Установите в HTTP-заголовке Authorization значение "Basic " + encoding, а затем введите содержимое страницы и выведите его на экран.

10. Используя регулярное выражение, извлеките все десятичные целые числа (в том числе и отрицательные) из символьной строки в списочный массив типа ArrayList<Integer>, используя, во-первых, метод find(), а во вторых — метод split(). Имейте в виду, что знак + или -, после которого не следует цифра, является разделителем.

11. Используя регулярные выражения, извлеките имена каталогов (в виде массива символьных строк), имя и расширение файла из абсолютного или относительного пути вроде /home/cay/myfile.txt.

12. Придумайте реальный пример применения ссылок на группы в методе Matcher.replaceAll() и реализуйте его на практике.

13. Реализуйте метод для получения клона любого объекта сериализируемого сначала в массив байтов и затем десериализируемого оттуда.

14. Реализуйте сериализируемый класс `Point` с переменными экземпляра для хранения координат точки *x* и *y*. Напишите одну программу для сериализации массива объектов типа `Point` в файл, а другую — для чтения из файла.

15. Продолжите предыдущее упражнение, но измените представление данных типа `Point` таким образом, чтобы хранить координаты точки в массиве. Что произойдет при попытке прочитать в новой версии файл, сформированный в предыдущей версии? И что произойдет, если исправить идентификатор `serialVersionUID`? Что бы вы сделали, если бы ваша жизнь зависела от создания новой версии, совместимой с прежней?

16. Какие классы из стандартной библиотеки Java реализуют интерфейс `Externalizable` и в каких из них применяются методы `writeReplace()` и `readResolve()`?

Глава 10

Параллельное программирование

В этой главе...

Java стал одним из первых среди основных языков программирования со встроенной поддержкой параллельного программирования. Уже первых программирующих на Java восхитила простота, с которой они могли загружать изображения в фоновых потоках исполнения или реализовывать веб-серверы, параллельно обслуживавшие многие запросы. В то время основное внимание уделялось постоянной загрузке центрального процессора (ЦП) тогда, как другие задачи ожидали доступа к сети. А ныне большинство компьютеров оснащено несколькими процессорами, и программисты должны позаботиться об их постоянной загрузке, чтобы они не простаивали без дела.

В этой главе речь пойдет о разбиении вычислений на параллельные задачи и надежном их исполнении. Основное внимание в ней уделяется потребностям прикладных, а не системных программистов, реализующих веб-серверы или разрабатывающих промежуточное программное обеспечение.

Именно поэтому материал этой главы организован таким образом, чтобы продемонстрировать сначала инструментальные средства, которые следует применять в практике параллельного программирования, а затем низкоуровневые конструкции, о которых полезно знать, чтобы оценить затраты на выполнение некоторых операций. Впрочем, программирование низкоуровневых потоков исполнения лучше предоставить опытным специалистам. Тем, кто желает стать одним из них, настоятельно рекомендуется прочитать отличную книгу *Java Concurrency in Practice* Брайана Гётца (Brian Goetz, издательство Addison-Wesley, 2006 г.).

Основные положения этой главы приведены ниже.

1. Интерфейс `Runnable` описывает задачу, которая может выполняться асинхронно.
2. Интерфейс `Executor` планирует для исполнения экземпляры задач типа `Runnable`.
3. Интерфейс `Callable` описывает задачу, которая выдает результат.
4. Один или несколько экземпляров задач типа `Callable` можно передать интерфейсу `ExecutorService`, чтобы объединить результаты, как только они станут доступны.
5. Если общие данные обрабатываются в нескольких потоках исполнения без синхронизации, то конечный результат непредсказуем.
6. Параллельным алгоритмам и потокобезопасным структурам данных следует отдавать предпочтение над программированием с блокировками.
7. Параллельные операции с потоками и массивами данных автоматически и надежно распараллеливают вычисления.
8. Класс `ConcurrentHashMap` представляет потокобезопасную хеш-таблицу, которая допускает атомарное обновление записей.
9. Для организации общего счетчика без блокировок можно воспользоваться классом `AtomicLong` или `LongAdder`, если сильно соперничество за такой счетчик.
10. Блокировка гарантирует одновременное выполнение критического раздела кода только в одном потоке.

11. Выполнение прерываемой задачи должно быть прекращено, если установлен признак прерывания или возникло исключение типа `InterruptedException`.
12. Выполнение длительной задачи не должно блокировать пользовательский интерфейс прикладной программы, но ход ее выполнения и окончательные обновления должны происходить в потоке исполнения пользовательского интерфейса.
13. Класс `Process` позволяет выполнить команду в отдельном процессе и взаимодействовать с потоками ввода, вывода данных и ошибок.

10.1. Параллельные задачи

При разработке параллельной программы нужно подумать о тех задачах, которые должны решаться параллельно. В последующих разделах поясняется, каким образом организуется параллельное выполнение задач.

10.1.1. Выполнение задач

В языке Java интерфейс `Runnable` описывает задачу, которую требуется выполнить (как правило, параллельно) с другими задачами. Ниже приведено определение интерфейса `Runnable`.

```
public interface Runnable {
  void run();
}
```

Код метода `run()` будет выполняться в *потоке исполнения*. Такой поток является механизмом исполнения последовательности инструкций, обычно предоставляемых операционной системой. Несколько потоков исполнения действуют параллельно, используя отдельные процессоры или разные интервалы времени на одном и том же процессоре.

Запустить поток исполнения можно только для данной задачи типа `Runnable`, как будет показано далее, в разделе 10.7.1. Но на практике не имеет смысла иметь взаимно однозначное соответствие задач и потоков исполнения. Если задачи краткосрочные, то многие из них можно выполнять в одном и том же потоке, чтобы не тратить зря время на запуск потока исполнения. А если задачи требуют интенсивных вычислений, то для каждого потока лучше выделить отдельный процессор вместо отдельного потока на каждую задачу, чтобы избежать издержек на переключение потоков исполнения.

В параллельной библиотеке Java *исполнитель* служит для исполнения задач, выбирая потоки, в которых они должны выполняться, как показано ниже.

```
Runnable task = () -> { ... };
Executor exec = ...;
exec.execute(task);
```

В классе `Executors` имеются фабричные методы для разных типов исполнителей. Так, в результате следующего вызова:

```
exec = Executors.newCachedThreadPool();
```

получается исполнитель, оптимизируемый для программ со многими задачами, которые являются краткосрочными или тратят бо́льшую часть своего времени на ожидание. Каждая задача выполняется в простаивающем потоке исполнения, если это возможно. А новый поток исполнения выделяется, если все существующие потоки исполнения уже заняты. Потоки исполнения, простаивающие продолжительное время, останавливаются.

А в результате следующего вызова:

```
exec = Executors.newFixedThreadPool(nthreads);
```

получается пул с фиксированным количеством потоков исполнения. Когда задача передается на выполнение, она ставится в очередь до тех пор, пока поток исполнения не становится доступным. Это удобно для решения задач, требующих интенсивных вычислений. Потоки исполнения можно породить, исходя из количества доступных процессоров, сведения о которых нетрудно получить следующим образом:

```
int processors = Runtime.getRuntime().availableProcessors();
```

Ниже приведен пример демонстрационной программы, в которой параллельно выполняются две задачи. Попробуйте выполнить эту программу, выбрав ее исходный код среди примеров кода, прилагаемого к данной книге.

```
public static void main(String[] args) {
  Runnable hellos = () -> {
    for (int i = 1; i <= 1000; i++)
      System.out.println("Hello " + i);
  };
  Runnable goodbyes = () -> {
    for (int i = 1; i <= 1000; i++)
      System.out.println("Goodbye " + i);
  };
  Executor executor = Executors.newCachedThreadPool();
  executor.execute(hellos);
  executor.execute(goodbyes);
}
```

Если выполнить эту программу несколько раз подряд, то результаты ее выполнения будут чередоваться, как показано ниже.

```
Goodbye 1
...
Goodbye 871
Goodbye 872
Hello 806
Goodbye 873
Goodbye 874
Goodbye 875
Goodbye 876
Goodbye 877
Goodbye 878
Goodbye 879
Goodbye 880
```

```
Goodbye 881
Hello 807
Goodbye 882
...
Hello 1000
```

НА ЗАМЕТКУ. Можно заметить, что данная программа какое-то время ожидает по окончании последнего вывода результата на экран. Она останавливается, когда сведенные в пул потоки исполнения долго простаивают, а исполнитель завершает их.

10.1.2. Службы исполнителей и будущих действий

Рассмотрим вычисление, разбиваемое на несколько подзадач, каждая из которых выдает частичный результат. По завершении всех задач требуется объединить результаты. Для решения подобных подзадач можно воспользоваться интерфейсом `Callable`. В отличие от метода `run()` из интерфейса `Runnable`, метод `call()` из интерфейса `Callable` возвращает значение, как показано ниже. А кроме того, метод `call()` может генерировать произвольные исключения.

```
public interface Callable<V> {
  V call() throws Exception;
}
```

Чтобы выполнить задачу типа `Callable`, потребуется экземпляр интерфейса `ExecutorService`, производного от интерфейса `Executor`. Такие объекты выдают методы `newCachedThreadPool()` и `newFixedThreadPool()` из класса `Executors`, как показано ниже.

```
ExecutorService exec = Executors.newFixedThreadPool();
Callable<V> task = ...;
Future<V> result = exec.submit(task);
```

Когда задача передается на выполнение, то в ответ получается *будущее действие* — объект, представляющий вычисление, результат которого будет доступен в некотором будущем времени. У интерфейса `Future` имеются следующие методы:

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;
boolean cancel(boolean mayInterruptIfRunning)
boolean isCancelled()
boolean isDone()
```

Метод `get()` блокируется до тех пор, пока не станет доступным результат или истечет время ожидания, после чего он возвратит вычисленное значение или сгенерирует исключение типа `ExecutionException`, заключающее в оболочку исключение, сгенерированное методом `call()`.

А метод `cancel()` пытается отменить задачу. Если задача еще не выполняется, то она не будет запланирована. В противном случае прерывается поток, исполняющий задачу, если параметр `mayInterruptIfRunning` принимает логическое значение `true`.

НА ЗАМЕТКУ. Если задачу требуется сделать прерываемой, то придется организовать периодические проверки запросов на прерывание следующим образом:

```
Callable<V> task = () -> {
  while (дополнительно выполняемые операции) {
    if (Thread.currentThread().isInterrupted()) return null;
      выполнить дополнительные операции
  }
  return result;
};
```

Это удобно для отмены любых задач по успешном завершении какой-нибудь другой подзадачи. Подробнее о прерывании потоков исполнения речь пойдет в разделе 10.7.2.

Как правило, задаче приходится ожидать результатов выполнения нескольких подзадач. Вместо того чтобы передавать на выполнение каждую подзадачу в отдельности, можно вызвать метод `invokeAll()`, передав ему коллекцию типа `Collection` экземпляров задач типа `Callable`.

Допустим, что требуется посчитать частоту, с которой слово встречается в ряде файлов. С этой целью создается объект типа `Callable<Integer>`, возвращающий подсчет для каждого файла. По завершении всех задач получается список будущих действий. А поскольку все они уже выполнены, то можно подытожить полученные результаты, как показано ниже.

```
String word = ...;
Set<Path> paths = ...;
List<Callable<Long>> tasks = new ArrayList<>();
for (Path p : paths) tasks.add(
    () -> { возвратить количество вхождений слова в p });
List<Future<Long>> results = executor.invokeAll(tasks);
long total = 0;
for (Future<Long> result : results) total += result.get();
```

Имеется также вариант метода `invokeAll()` с установкой времени ожидания. В этом варианте отменяются все задачи, которые еще не были выполнены по истечении времени ожидания.

НА ЗАМЕТКУ. Если вас не устраивает, что вызывающая задача блокируется до тех пор, пока все подзадачи не будут завершены, воспользуйтесь интерфейсом **`ExecutorCompletionService`**. Он возвращает будущие действия по порядку их завершения, как показано ниже.

```
ExecutorCompletionService service =
    new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++) {
  обработать результат вызова service.take().get()
  сделать что-нибудь еще
}
```

Метод `invokeAny()` действует аналогично методу `invokeAll()`, но возвращает результат, как только любая из переданных на выполнение задач завершится без генерирования исключения. И тогда он возвращает значение своего объекта типа `Future`,

тогда как остальные задачи отменяются. Это удобно для поиска, который можно завершить, как только будет обнаружено совпадение. Так, в следующем примере кода обнаруживается файл, содержащий заданное слово:

```
String word = ...;
Set<Path> files = ...;
List<Callable<Path>> tasks = new ArrayList<>();
for (Path p : files) tasks.add(
    () -> { if (слово встречается в p) return p; else throw ... });
Path found = executor.invokeAny(tasks);
```

Как видите, большую часть хлопот берет на себя интерфейс `ExecutorService`. Он не только назначает задачи для потоков исполнения, но и организует результаты выполнения задач, исключения и отмену задач.

10.2. Безопасность потоков исполнения

Многие программисты первоначально считают, что освоить параллельное программирование совсем не трудно. Достаточно только разбить выполняемую работу на задачи. Что же может пойти далее не так, как предполагалось? В последующих разделах поясняется, что именно может пойти не так, а также дается краткий обзор того, как с этим справиться.

10.2.1. Доступность

Даже такие простые операции, как запись и чтение переменной, могут невероятно усложниться при выполнении на современных процессорах. Рассмотрим следующий пример:

```
private static boolean done = false;

public static void main(String[] args) {
  Runnable hellos = () -> {
    for (int i = 1; i <= 1000; i++)
      System.out.println("Hello " + i);
    done = true;
  };
  Runnable goodbye = () -> {
    int i = 1;
    while (!done) i++;
    System.out.println("Goodbye " + i);
  };
  Executor executor = Executors.newCachedThreadPool();
  executor.execute(hellos);
  executor.execute(goodbye);
}
```

В первой задаче слово "Hello" выводится тысячу раз, а затем переменной `done` присваивается логическое значение `true`. А вторая задача ожидает до тех пор, пока переменной не будет присвоено логическое значение `true`, после чего слово

"Goodbye" выводится один раз, а счетчик инкрементируется в ожидании этого счастливого момента. В итоге можно ожидать результат, аналогичный следующему:

```
Hello 1
...
Hello 1000
Goodbye 501249
```

Когда я попытался выполнить программу из данного примера на своем переносном компьютере с двухъядерным процессором, она вывела на экран сообщение "Hello 1000", но так и не завершилась. Результат следующего присваивания:

```
done = true;
```

оказался *недоступным* для потока, исполнявшего вторую задачу. А почему он оказался недоступным? На то имеется несколько причин, которые могут быть связаны с *кешированием* и *переупорядочением инструкций*.

Для переменной done в оперативной памяти компьютера выделяется определенное место. Но оперативная память действует намного медленнее современных процессоров. Поэтому процессор пытается сохранить нужные ему данные в своих регистрах или во встроенной сверхоперативной памяти, записывая окончательные изменения в основной оперативной памяти. Такое кеширование данных просто необходимо для производительной работы процессора. Для синхронизации кешируемых копий данных имеются специальные операции, но они выполняются только по запросу.

Но и это еще не все. Компилятор, виртуальная машина и процессор могут изменить порядок выполнения инструкций ради ускорения операций, при условии, что это не изменит семантику выполняемой программы.

Рассмотрим в качестве примера следующие вычисления:

```
x = Что-нибудь без учета координаты y;
y = Что-нибудь без учета координаты x;
z = x + y;
```

Две первые стадии вычислений должны непременно произойти до третьей стадии, но могут быть выполнены в любом порядке. Процессор может выполнить две первые стадии параллельно (и зачастую так и происходит) или поменять порядок их выполнения, если исходные данные для второй стадии окажутся доступными раньше.

В рассматриваемом здесь примере следующий цикл:

```
while (!done) i++;
```

может быть переупорядочен таким образом:

```
if (!done) while (true) i++;
```

поскольку в теле этого цикла значение переменной done не изменяется.

По умолчанию при оптимизации предполагается, что параллельный доступ к оперативной памяти отсутствует. Если же такой доступ имеется, то о нем должно

быть известно виртуальной машине, чтобы она могла выдать процессору инструкции, препятствующие неверному переупорядочению.

Доступность общей переменной для обновления можно обеспечить несколькими способами. Все эти способы перечислены ниже.

1. Значение конечной переменной, объявленной как `final`, доступно после ее инициализации.
2. Исходное значение статической переменной, объявленной как `static`, доступно после ее инициализации.
3. Изменения в изменчивой переменной, объявленной как `volatile`, доступны сразу.
4. Изменения, происходящие до освобождения блокировки, доступны в любом коде, получающем ту же самую блокировку (подробнее от этом речь пойдет в разделе 10.6.1).

В данном случае затруднение устраняется, если общую переменную `done`, разделяемую двумя задачами, объявить с модификатором `volatile` следующим образом:

```
private static volatile boolean done;
```

В таком случае компилятор сформирует нужные инструкции, чтобы любые изменения переменной `done`, произведенные в одной задаче, были доступны другим задачам. Для разрешения данного затруднения оказалось достаточно модификатора доступа `volatile`, но объявление общих переменных как `volatile` нельзя считать решением, пригодным на все случаи жизни, как станет ясно из последующего раздела.

СОВЕТ. Любое поле, которое не изменяется после инициализации, рекомендуется объявить как **`final`**, чтобы больше не беспокоиться о его доступности.

10.2.2. Состояние гонок

Допустим, что в нескольких параллельно выполняющихся задачах обновляется общий разделяемый ими целочисленный счетчик:

```
private static volatile int count = 0;
...
count++; // Задача 1
...
count++; // Задача 2
...
```

Переменная `count` объявлена как `volatile`, а следовательно, ее обновления доступны всем задачам, но этого недостаточно. Ведь обновление `count++` не является *атомарным*. По существу, оно означает следующее:

```
count = count + 1;
```

и может быть прервано, если поток исполнения вытесняется до сохранения в нем значения count + 1 обратно в переменной count. Рассмотрим следующий случай:

```
int count = 0; // Исходное значение
register1 = count + 1; // В потоке 1 вычисляется значение count + 1
... // Поток 1 вытесняется
register2 = count + 1; // В потоке 2 вычисляется значение count + 1
count = register2; // В потоке 2 значение 1 сохраняется в переменной count
... // Поток 1 снова исполняется
count = register1; // В потоке 1 значение 1 сохраняется в переменной count
```

В итоге переменная count содержит значение 1, а не 2. Такого рода ошибка называется *состоянием гонок*, поскольку оно зависит от того, какой именно поток исполнения одержит верх в "гонке" за обновление общей переменной.

Насколько реальна подобная ошибка? Она вполне реальна. Чтобы убедиться в этом, выполните приведенную ниже демонстрационную программу из примеров исходного кода, прилагаемого к данной книге. В этой программе организуется 100 потоков исполнения, в каждом из которых счетчик инкрементируется 1000 раз и выводится полученный результат.

```
for (int i = 1; i <= 100; i++) {
  int taskId = i;
  Runnable task = () -> {
    for (int k = 1; k <= 1000; k++)
      count++;
    System.out.println(taskId + ": " + count);
  };
  executor.execute(task);
}
```

Вывод результатов выполнения этой программы обычно начинается без особых осложнений и выглядит так, как показано ниже.

```
1: 1000
3: 2000
2: 3000
6: 4000
```

А некоторое время спустя он уже настораживает:

```
72: 58196
68: 59196
73: 61196
71: 60196
69: 62196
```

Причина заключается в том, что некоторые потоки исполнения могли быть приостановлены в самый неподходящий момент. Но самое главное — что же происходит с задачей, которая завершается последней? Довела ли она значение счетчика до 100000?

Я выполнил эту программу десятки раз на своем переносном компьютере с двухъядерным процессором, и всякий раз она давала сбои. В прошлом, когда персональные компьютеры были оснащены единственным ЦП, наблюдать состояние гонок было труднее, и поэтому программисты не замечали столь значительные сбои в работе параллельных программ. Но ведь совершенно не важно, вычисляется ли неверное значение в течение секунд или часов.

Рассмотренный выше пример служит упрощенным вариантом применения общего счетчика в игровой программе. Аналогичное затруднение демонстрируется на реальном примере в упражнении 16 в конце этой главы. Но состояние гонок возникает не только при доступе к счетчикам, а вообще к любым общим переменным, которые изменяются в ходе параллельных вычислений. Например, код для постановки значения в голову очереди может выглядеть следующим образом:

```
Node n = new Node();
if (head == null) head = n;
else tail.next = n;
tail = n;
tail.value = newValue;
```

Если выполнение сложной последовательности инструкций приостановится в самый неподходящий момент и другая задача получит управление для доступа к очереди, когда она находится в неустойчивом состояния, то многое может пойти не так, как предполагалось. Выполните упражнение 20 в конце этой главы, чтобы получить некоторое представление о том, как можно испортить структуру данных в результате ее изменения в параллельном режиме работы программы.

В качестве действенного средства борьбы с состоянием гонок служат *блокировки*, позволяющие сделать атомарными ответственные последовательности операций. О параллельном программировании с помощью блокировок речь пойдет в разделе 10.6.1. Но, к сожалению, блокировки разрешают далеко не все затруднения, возникающие при параллельном программировании. Кроме того, пользоваться ими непросто и легко совершить ошибки, существенно снижающие производительность и даже доводящие до состояния взаимной блокировки.

10.2.3. Методики надежного параллельного программирования

Программирующим на таких языках, как C и C++, приходится вручную выделять и освобождать оперативную память, а это опасно. Многие программисты потратили впустую немало времени на устранение ошибок выделения и освобождения оперативной памяти в своих программах. А в Java для этой цели имеется сборщик "мусора", и лишь немногим программирующим на этом языке требуется решать задачи управления памятью.

К сожалению, аналогичный механизм доступа к общим данным в параллельной программе отсутствует. Поэтому остается только следовать рекомендациям по предупреждению опасностей, присущих параллельному доступу к общим данным. Ниже рассматриваются некоторые методики надежного параллельного программирования.

Весьма эффективной оказывается методика *привязки* к потоку исполнения. Для этого достаточно отказаться от разделения общих данных среди параллельно

выполняемых задач. Так, если в нескольких задачах требуется что-нибудь подсчитывать, то для каждой из них следует назначить свой закрытый счетчик, вместо того чтобы обновлять общий счетчик. А по завершении задач полученные результаты можно передать еще одной задаче, которая подытожит их.

Еще одна полезная методика подразумевает *неизменяемость* объектов. Неизменяемыми объектами можно благополучно обмениваться или делать их общими. Например, вместо того чтобы вводить результаты в общую коллекцию, в отдельной задаче можно сформировать неизменяемую коллекцию результатов, а в другой задаче — подытожить результаты в неизменяемой структуре данных. Общий замысел такой методики довольно прост, но ее реализация на практике имеет свои особенности, которые следует иметь в виду (подробнее об этом — далее, в разделе 10.2.4).

Третья методика основывается на *блокировке*. Предоставляя задачам доступ к структуре данных по очереди, можно избежать ее порчи. В разделе 10.4 будут рассмотрены структуры данных, предоставляемые параллельной библиотекой. Ими можно благополучно пользоваться в параллельном программировании. А в разделе 10.6.1 будет показано, как строить подобные структуры данных, применяя механизм блокировки.

Блокировка может оказаться затратной, поскольку она снижает возможности распараллеливания задач. И если имеется немало задач, вносящих результаты в общую хеш-таблицу, которая блокируется при каждом обновлении, то блокировка становится узким местом. Если большинству задач приходится ожидать своей очереди, пользы от них немного. Иногда оказывается возможным *разделение* данных таким образом, чтобы разные их фрагменты были доступны параллельно. Такое разделение применяется в ряде структур данных из параллельной библиотеки Java, а также в параллельных алгоритмах из библиотеки потоков данных. Но разделение данных требует умелого обращения и немалого труда для достижения нужного результата. Поэтому вместо него лучше пользоваться структурами данных и алгоритмами из библиотеки Java.

10.2.4. Неизменяемые классы

Класс считается неизменяемым, если построенные однажды его экземпляры не подлежат дальнейшим изменениям. На первый взгляд, с ними ничего особенного нельзя сделать, но на самом деле это не совсем так. Например, в главе 12 будет показано, как обращаться с неизменяемыми объектами из библиотеки Java для даты и времени. Каждый экземпляр даты является неизменяемым, тем не менее, новые даты можно получить, например, дату следующего дня после заданного.

В качестве другого примера рассмотрим накопление результатов. Для этой цели можно было бы воспользоваться изменяемой коллекцией типа `HashSet`, обновляя ее так, как показано ниже. Но, очевидно, что это опасно.

```
results.addAll(newResults);
```

Неизменяемое множество всегда создает новые множества. Накапливаемые результаты можно, например, обновлять следующим образом:

```
results = results.union(newResults);
```

Изменение по-прежнему происходит, но ведь намного проще контролировать то, что происходит в одной переменной, чем в целом хеш-множестве, да еще и многими методами. Реализовать неизменяемые классы нетрудно, но при этом нужно обратить внимание на следующее.

1. Переменные экземпляра должны обязательно объявляться как `final`. И нет никаких причин не делать этого, кроме получения важного преимущества. Виртуальная машина гарантирует доступность конечной переменной экземпляра после ее создания (см. раздел 10.2.1).
2. Разумеется, ни один из методов не должен быть модифицирующим. Их можно объявить как `final` или сделать то же самое с их классом, чтобы эти методы нельзя было переопределить модифицирующими методами.
3. Нельзя допускать утечки изменяемого состояния. Ни один из (незакрытых) методов не должен возвращать ссылку на свои внутренние элементы, которые могут быть изменены. А если в методе из одного класса вызывается метод из другого класса, то никаких подобных ссылок передавать ему нельзя, поскольку в вызываемом методе они могут быть использованы для нежелательного изменения. Возвратить или передать можно только копию, если в этом есть потребность.
4. Ссылка `this` не должна исчезать в конструкторе. Если вызывается другой метод, то ему нельзя передавать внутренние ссылки. А как насчет ссылки `this`? Ее вполне допустимо передавать после построения объекта. Но если она обнаружена в конструкторе, то вполне возможно, то кто-нибудь наблюдал объект в незавершенном состоянии.

НА ЗАМЕТКУ. Соблюсти первые три из приведенных выше рекомендаций нетрудно, тогда как последняя рекомендация кажется довольно формальной и нетипичной. Примерами тому служат запуск потока исполнения в конструкторе, где интерфейс **`Runnable`** реализуется внутренним классом, содержащим ссылку **`this`**, или ввод ссылки **`this`** в очередь приемников при построении приемника событий. Подобные действия лучше выполнять по завершении построения.

10.3. Параллельные алгоритмы

Прежде чем приступать к распараллеливанию вычислений, следует проверить, сделала ли уже это библиотека Java автоматически. Библиотека потоков данных или класс `Arrays` может взять подобные хлопоты на себя.

10.3.1. Параллельные потоки данных

Библиотека потоков данных автоматически распараллеливает операции в крупных потоках данных. Так, если `coll` является крупной коллекцией символьных строк и требуется выяснить, сколько таких строк начинается с буквы **A**, то для этой цели достаточно сделать следующий вызов:

```
long result = coll.parallelStream().filter(s -> s.startsWith("A")).count();
```

Метод parallelStream() возвращает параллельный поток данных, который разбивается на сегменты. Фильтрация и подсчет выполняются над каждым сегментом в отдельности, а полученные результаты затем подытоживаются. Таким образом, беспокоиться о подробностях процесса обработки данных в параллельном потоке не нужно.

В качестве другого примера допустим, что требуется подсчитать частоту, с которой заданное слово встречается во всех подкаталогах отдельного каталога. Пути к ним можно получить в виде потока данных следующим образом:

```
try (Stream<Path> paths = Files.walk(pathToRoot)) {
   ...
}
```

Полученный поток данных достаточно превратить в параллельный, а затем вычислить сумму полученных подсчетов, как показано ниже.

```
int total = paths.parallel()
    .mapToInt(p -> { return количество вхождений слова в p })
    .sum();
```

ВНИМАНИЕ. Если параллельные потоки данных употребляются вместе с лямбда-выражениями (в виде аргумента методов **filter()** и **mapToInt()** в предыдущих примерах кода), то следует избегать ненадежного изменения общих объектов.

10.3.2. Параллельные операции с массивами

В классе Arrays поддерживается целый ряд распараллеливаемых операций. Аналогично рассмотренным выше операциям с параллельными потоками данных, операции с массивом разбивают его на части, обрабатывают их параллельно и подытоживают полученные результаты.

Статический метод Arrays.parallelSetAll() заполняет массив значениями, вычисляемыми функцией. Эта функция получает индекс элемента массива и вычисляет значение по данному индексу, как показано ниже.

```
Arrays.parallelSetAll(values, i -> i % 10);
  // Заполняет массив значениями 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```

Очевидно, что такая операция только выигрывает от распараллеливания. Имеются ее разновидности для массивов всех примитивных типов и массивов объектов.

Метод parallelSort() позволяет отсортировать массив примитивных значений или объектов, как в следующей строке кода:

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

Всем подобным методам можно предоставить пределы диапазона следующим образом:

```
Arrays.parallelSort(values, values.length / 2, values.length);
    // отсортировать верхнюю половину заданного диапазона значений
```

НА ЗАМЕТКУ. На первый взгляд кажется несколько странным, что упомянутые выше методы имеют слово **parallel** в своих именах. Ведь пользователя вообще не интересует, каким образом выполняются операции установки и сортировки значений в массиве. Но разработчики прикладного программного интерфейса Java API стремились к тому, чтобы из наименования подобных операций пользователям было ясно, что они распараллеливаются. Подобным образом пользователи предупреждаются о недопустимости передачи функций с побочными эффектами.

И наконец, имеется специальный метод `parallelPrefix()`, пример применения которого приводится в упражнении 4 в конце данной главы. А для выполнения других параллельных операций с массивами их нужно сначала превратить в потоки данных. Например, чтобы вычислить сумму целых значений в длинном массиве, достаточно сделать следующий вызов:

```
long sum = IntStream.of(values).parallel().sum();
```

10.4. Потокобезопасные структуры данных

Если структура данных (например, очередь или хеш-таблица) параллельно изменяется в нескольких потоках исполнения, то ее внутреннее строение можно легко нарушить. Так, в одном потоке исполнения может начаться ввод нового элемента, но посредине перенаправления ссылок он вытесняется другим потоком исполнения, который, в свою очередь, начинает обход структуры данных с того же самого места. В таком случае во втором потоке будет совершен переход по недостоверным ссылкам, что приведет структуру данных в полный беспорядок с возможным генерированием исключений или даже входом в бесконечный цикл.

Как будет показано в разделе 10.6.1, для гарантии одновременного доступа к структуре данных только из одного потока исполнения можно заблокировать все остальные потоки исполнения, хотя это и неэффективно. Коллекции из пакета `java.util.concurrent` были предусмотрительно реализованы таким образом, чтобы несколько потоков исполнения имели доступ к ним, не блокируя друг друга, но при условии, что они получают доступ к разным частям коллекции.

НА ЗАМЕТКУ. Такие коллекции предоставляют *слабо совместные итераторы*. Это означает, что итераторы представляют элементы, которые появляются в начале итерации, но могут и не отражать некоторые или все изменения, внесенные после построения этих итераторов. Тем не менее такой итератор не генерирует исключение типа **ConcurrentModificationException**.

С другой стороны, итератор коллекции из пакета **java.util** генерирует исключение типа **ConcurrentModificationException**, если коллекция изменяется после построения ее итератора.

10.4.1. Параллельные хеш-отображения

Прежде всего, класс `ConcurrentHashMap` представляет хеш-отображение, операции над которым являются потокобезопасными. Независимо от того, сколько потоков исполнения оперируют отображением одновременно, его внутреннее строение не будет

нарушено. Разумеется, некоторые потоки исполнения могут быть временно заблокированы, но отображение может эффективно поддерживать большое количество параллельных средств чтения и определенное количество параллельных средств записи.

Но этого недостаточно. Допустим, что требуется воспользоваться отображением для подсчета частоты появления некоторых слов в нескольких потоках исполнения. Очевидно, что приведенный ниже код для обновления подсчета не является потокобезопасным. Ведь тот же самый подсчет может быть одновременно обновлен в другом потоке исполнения.

```
ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();
...
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // ОШИБКА: заменить значение
                         // переменной oldValue нельзя!
```

Чтобы надежно обновить значение, следует воспользоваться методом `compute()`. Этот метод вызывается с ключом и функцией для вычисления нового значения. Данная функция получает ключ и связанное с ним значение, а если таковое отсутствует, то пустое значение `null`, и далее вычисляет новое значение. Так, в следующей строке кода показано, как правильно обновлять подсчет слов в рассматриваемом здесь примере:

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```

Метод `compute()` выполняет *атомарную* операцию. Это означает, что ни в одном другом потоке исполнения нельзя изменить запись в отображении до тех пор, пока не завершится вычисление в текущем потоке. Имеются варианты `computeIfPresent()` и `computeIfAbsent()` данного метода, вычисляющие новое значение только в том случае, если уже имеется или еще отсутствует прежнее значение.

Еще одну атомарную операцию выполняет метод `putIfAbsent()`. С его помощью счетчик может быть инициализирован следующим образом:

```
map.putIfAbsent(word, 0L);
```

Когда ключ вводится в первый раз, нередко требуется сделать что-нибудь особенное. Очень удобным для этой цели является метод `merge()`. Он принимает в качестве параметра начальное значение, используемое в отсутствие ключа. В противном случае вызывается предоставляемая функция, объединяющая существующее значение с новым. (В отличие от метода `compute()`, эта функция не обрабатывает ключ.) Метод `merge()` можно вызвать следующим образом:

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue + newValue);
```

или просто так:

```
map.merge(word, 1L, Long::sum);
```

Разумеется, функции, передаваемые методам `compute()` и `merge()`, должны выполняться быстро. Кроме того, они не должны и пытаться изменить отображение.

НА ЗАМЕТКУ. Имеются методы, автоматически удаляющие или заменяющие запись, если в настоящий момент она равна уже существующей записи. До внедрения метода **compute()** для инкрементирования подсчета программирующим на Java приходилось писать код, аналогичный следующему:

```
do {
   oldValue = map.get(word);
   newValue = oldValue + 1;
} while (!map.replace(word, oldValue, newValue));
```

НА ЗАМЕТКУ. Имеется ряд *групповых операций* для поиска, преобразования или обхода отображения типа **ConcurrentHashMap**. Они оперируют моментальным снимком данных и могут надежно выполняться даже в тот момент, когда отображением оперируют другие потоки исполнения. В документации на прикладной программный интерфейс API можно найти описание операций, имена которых начинаются на **search**, **reduce** и **forEach**. Имеются их разновидности, оперирующие ключами, значениями и записями. Операции типа **reduce** различаются по функциям сведения значений типа **int**, **long** и **double**.

10.4.2. Блокирующие очереди

К числу наиболее употребительных средств для согласования параллельного выполнения задач относится *блокирующая очередь*. Задачи поставщиков вводят элементы в очередь, а задачи потребителей извлекают их из очереди. Такая очередь позволяет надежно передавать данные от одной задачи к другой.

При попытке ввести элемент в уже заполненную очередь или удалить элемент из пустой очереди операция блокируется. Подобным образом очередь выравнивает рабочую нагрузку. Если задачи поставщиков выполняются медленнее, чем задачи потребителей, последние блокируются, ожидая результатов. А если задачи поставщиков выполняются быстрее, то очередь заполняется до тех пор, пока потребители не ликвидируют отставание.

В табл. 10.1 перечислены методы для блокирующих очередей. Эти методы делятся на три категории по действиям, выполняемым, когда очередь заполнена или пуста. Помимо блокирующих методов, имеются методы, генерирующие исключение при неудачном исходе операции, а также методы, возвращающие признак сбоя вместо того, чтобы генерировать исключение, если они не в состоянии выполнить свои задачи.

Таблица 10.1. Операции с блокирующим очередями

Метод	Обычное действие	Действие при ошибке
put()	Вводит элемент в хвост очереди	Блокирует операцию, если очередь заполнена
take()	Удаляет элемент из головы очереди и возвращает его	Блокирует операцию, если очередь пуста
add()	Вводит элемент в хвост очереди	Генерирует исключение типа **IllegalStateException**, если очередь заполнена
remove()	Удаляет элемент из головы очереди и возвращает его	Генерирует исключение типа **NoSuchElementException**, если очередь пуста

Окончание табл. 10.1

Метод	Обычное действие	Действие при ошибке
`element()`	Возвращает элемент из головы очереди	Генерирует исключение типа `NoSuchElementException`, если очередь пуста
`offer()`	Вводит элемент в очередь и возвращает логическое значение `true`	Возвращает логическое значение `false`, если очередь заполнена
`poll()`	Удаляет элемент из головы очереди и возвращает его	Возвращает пустое значение `null`, если очередь пуста
`peek()`	Возвращает элемент из головы очереди	Возвращает пустое значение `null`, если очередь пуста

НА ЗАМЕТКУ. Методы `poll()` и `peek()` возвращают пустое значение `null` для обозначения сбоя. Следовательно, пустые значения вполне допустимо вводить в блокирующие очереди.

Имеются также варианты методов `offer()` и `poll()` с установкой времени ожидания. Например, при следующем вызове:

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

в течение 100 миллисекунд предпринимается попытка ввести элемент в хвост очереди. Если эта попытка окажется успешной, то возвращается логическое значение `true`, а по истечении времени ожидания — логическое значение `false`. Аналогично при следующем вызове:

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

в течение 100 миллисекунд предпринимается попытка удалить элемент из головы очереди. Если эта попытка окажется успешной, то возвращается элемент из головы очереди, а по истечении времени ожидания — пустое значение `null`.

В пакете `java.util.concurrent` предоставляется несколько вариантов блокирующих очередей. В частности, класс `LinkedBlockingQueue` опирается на связный список, а в классе `ArrayBlockingQueue` применяется циклический массив.

В упражнении 10, что в конце этой главы, показано, как пользоваться блокирующими очередями для анализа файлов в каталогах. В одном потоке исполнения осуществляется обход дерева каталогов и ввод файлов в очередь, а в ряде других потоков исполнения — поиск и удаление файлов. При таком применении блокирующей очереди вполне вероятно, что поставщик быстро заполнит очередь файлами и заблокируется до тех пор, пока потребители не ликвидируют отставание.

Типичный недостаток такой конструкции заключается в остановке потребителей. Если очередь пуста, потребитель просто не сможет завершить свою задачу, возможно, потому, что задача поставщика еще не начата или запоздала. Если имеется единственный поставщик, он может ввести в очередь признак последнего элемента аналогично фиктивному чемодану с меткой "Последний багаж" на ленте транспортера в зале выдачи багажа в аэропорту.

10.4.3. Другие потокобезопасные структуры данных

Аналогично выбору между хеш-отображениями и древовидными отображениями из пакета `java.util` можно выбрать параллельное отображение типа `ConcurrentSkipListMap`, опирающееся на сравнение ключей. Это отображение применяется в том случае, если требуется обойти ключи в сортированном порядке или воспользоваться одним из методов, добавленных в интерфейс `NavigableMap` (см. главу 7). То же самое можно сказать и о параллельном множестве типа `ConcurrentSkipListSet`.

Потокобезопасными являются также коллекции типа `CopyOnWriteArrayList` и `CopyOnWriteArraySet`, в которых все модифицирующие методы создают копию массива, положенного в основу коллекции. Такая организация удобна, если количество потоков исполнения, перебирающих коллекцию, заметно превышает количество потоков исполнения, которые изменяют ее. Когда строится итератор, он получает ссылку на текущий массив. Если в дальнейшем этот массив изменяется, итератор по-прежнему имеет прежний массив, но массив коллекции заменяется. Следовательно, прежний итератор имеет согласованное (но потенциально устаревшее) представление, доступное ему без всяких издержек на синхронизацию.

Допустим, что требуется крупное, потокобезопасное множество вместо отображения. Для этой цели отсутствует класс `ConcurrentHashSet`, но пытаться создавать его самостоятельно вряд ли стоит. Разумеется, можно воспользоваться классом `ConcurrentHashMap` с поддельными значениями. Но в итоге получается отображение, а не множество, над которым нельзя выполнять операции из интерфейса `Set`.

Приведенный ниже статический метод `newKeySet()` производит множество типа `Set<K>`, которое на самом деле является оболочкой для отображения типа `ConcurrentHashMap<K, Boolean>`. (Все значения в этом отображении являются логическими типа `Boolean.TRUE`. Впрочем, это совсем не важно, поскольку данное отображение употребляется лишь как множество.)

```
Set<String> words = ConcurrentHashMap.newKeySet();
```

Если уже имеется отображение, то метод `keySet()` производит из него множество ключей, которое является изменяемым. Так, если удалить элементы из этого множества, ключи (и их значения) удаляются из исходного отображения. Но вводить элементы в множество ключей вряд ли стоит, поскольку для ввода отсутствуют соответствующие значения. Метод `keySet()` можно еще раз вызвать со значением по умолчанию, используемым для ввода элементов в множество, как показано ниже. Если элемент `"Java"` прежде отсутствовал в множестве `words`, то теперь он имеет значение `1`.

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

10.5. Атомарные значения

Если общий счетчик обновляется в нескольких потоках исполнения, то нужно обеспечить потокобезопасный способ выполнения этой операции. В пакете `java.util.`

concurrent.atomic имеется целый ряд классов, пользующихся надежными и эффективными инструкциями машинного уровня для гарантии атомарности операций над значениями типа int, long и boolean, ссылками на объекты и их массивами. В общем, для принятия решения, пользоваться ли атомарными значениями вместо блокировок, требуется значительный опыт. Но атомарные счетчики и накапливающие сумматоры очень удобны для прикладного программирования.

Например, последовательность чисел можно надежно сформировать следующим образом:

```
public static AtomicLong nextNumber = new AtomicLong();
// В некотором потоке исполнения . . .
long id = nextNumber.incrementAndGet();
```

Метод incrementAndGet() атомарно инкрементирует значение типа AtomicLong и возвращает значение после инкремента. Это означает, что операции получения значения, добавления 1 (т.е. инкремента), установки и получения нового значения не могут быть прерваны. Этим гарантируется вычисление и возврат правильного значения, даже если из нескольких потоков исполнения осуществляется параллельный доступ к одному и тому же экземпляру.

Имеются также методы для атомарных операций установки, сложения и вычитания значений. Но что, если требуется более сложное обновление значений? С этой целью можно, в частности, воспользоваться методом updateAndGet(). Допустим, что требуется отследить наибольшее значение, наблюдающееся в разных потоках исполнения. Следующий код окажется неработоспособным:

```
public static AtomicLong largest = new AtomicLong();
// В некотором потоке исполнения . . .
largest.set(Math.max(largest.get(), observed));
    // ОШИКА: состояние гонок!
```

Такое обновление не является атомарным. Вместо этого лучше вызвать метод updateAndGet() с лямбда-выражением, чтобы обновить значение переменной. В данном примере можно сделать следующий вызов:

```
largest.updateAndGet(x -> Math.max(x, observed));
```

или такой вызов:

```
largest.accumulateAndGet(observed, Math::max);
```

Метод accumulateAndGet() принимает в качестве второго аргумента двоичную операцию, которая служит для объединения атомарного значения с первым аргументом. Имеются также методы getAndUpdate() и getAndAccumulate(), возвращающие прежнее значение.

НА ЗАМЕТКУ. Упомянутые выше методы предоставляются и для классов **AtomicInteger**, **AtomicIntegerArray**, **AtomicIntegerFieldUpdater**, **AtomicLongArray**, **AtomicLongFieldUpdater**, **AtomicReference**, **AtomicReferenceArray** и **AtomicReferenceFieldUpdater**.

При наличии очень большого количества потоков исполнения, получающих доступ к одним и тем же атомарным значениям, производительность снижается, поскольку обновления выполняются *оптимистично*. Это означает, что в ходе операции новое значение сначала вычисляется из заданного прежнего значения, а затем производится замена, при условии, что прежнее значение все еще остается текущим, а иначе попытка повторяется. При сильном соперничестве за доступ обновления требуют слишком много попыток.

Это затруднение разрешают классы `LongAdder` и `LongAccumulator` для некоторых типичных обновлений. В частности, класс `LongAdder` состоит из нескольких переменных, общая сумма значений которых составляет текущее значение. Отдельные слагаемые этого значения могут обновляться в нескольких потоках исполнения, а новые слагаемые — автоматически предоставляться при увеличении количества потоков исполнения. Это эффективно в общем случае, когда значение является суммарным и не требуется до тех пор, пока не будет завершена операция в целом. Благодаря этому заметно повышается производительность (см. упражнение 8 в конце главы).

Если предвидится сильное соперничество за доступ, то вместо класса `AtomicLong` следует воспользоваться классом `LongAdder`. Имена методов в этом классе несколько иные. Так, для инкремента счетчика следует вызвать метод `increment()`, для добавления заданной величины — метод `add()`, а для получения итоговой суммы — метод `sum()`, как показано ниже.

```
final LongAdder count = new LongAdder();
for (...)
  executor.execute(() -> {
    while (...) {
      ...
      if (...) count.increment();
    }
  });
...
long total = count.sum();
```

НА ЗАМЕТКУ. Разумеется, метод **`increment()`** *не* возвращает прежнее значение. Ведь это свело бы на нет весь выигрыш в эффективности от разбиения суммы на несколько слагаемых.

Класс `LongAccumulator` обобщает данный принцип до произвольной операции накопления. В качестве параметров конструктору этого класса предоставляется сама операция и ее нейтральный элемент. Для внедрения новых значений вызывается метод `accumulate()`, а для получения текущего значения — метод `get()`:

```
LongAccumulator accumulator = new LongAccumulator(Long::sum, 0);
// В некоторых задачах . . .
accumulator.accumulate(value);
// Когда вся операция завершена
long sum = accumulator.get();
```

В накапливающем сумматоре имеются переменные a_1, a_2, ..., a_n. Каждая переменная инициализируется нейтральным элементом (в данном случае — 0).

Когда метод `accumulate()` вызывается со значением **v**, одна из этих переменных автоматически обновляется следующим образом: $a_i = a_i$ *op* **v**, где *op* — операция накопления в инфиксной форме записи. В данном примере в результате вызова метода `accumulate()` вычисляется сумма $a_i = a_i + v$ для некоторой величины **i**.

А вызов метода `get()` приводит к такому результату: a_1 *op* a_2 *op* ... *op* a_n. В данном примере это сумма всех накапливающих сумматоров $a_1 + a_2 + op \ldots + a_n$.

Если выбрать другую операцию, то можно вычислить максимум или минимум (см. упражнение 9 в конце этой главы). В общем, операция должна быть ассоциативной или коммуникативной. Это означает, что конечный результат не должен зависеть от порядка, в котором объединяются промежуточные значения.

Имеются также классы `DoubleAdder` и `DoubleAccumulator`. Они действуют аналогичным образом, только оперируют значениями типа `double`.

СОВЕТ. Если пользоваться хеш-отображением типа **`LongAdder`**, то для инкремента сумматора по ключу можно применить следующий подход:

```
ConcurrentHashMap<String,LongAdder> counts = ...;
counts.computeIfAbsent(key, k -> new LongAdder()).increment();
```

Когда подсчет по ключу **`key`** инкрементируется первый раз, устанавливается новый сумматор.

10.6. Блокировки

А теперь, когда рассмотрен ряд средств, которыми прикладные программисты могут надежно пользоваться для структуризации параллельных прикладных программ, возникает вопрос: как построить потокобезопасный счетчик или блокирующую очередь? В последующих разделах поясняется, как это делается, чтобы стали понятны затраты и сложности данного процесса.

10.6.1. Реентерабельные блокировки

Чтобы избежать порчи общих переменных, нужно обеспечить возможность вычисления и установки новых значений одновременно только в одном потоке исполнения. Код, который должен быть выполнен полностью и без прерывания, называется *критическим разделом*. Для реализации критического раздела можно воспользоваться *блокировкой* следующим образом:

```
Lock countLock = new ReentrantLock(); // Общая блокировка счетчика
                                      // для нескольких потоков исполнения
int count; // Общий счетчик для нескольких потоков исполнения
...
countLock.lock();
try {
  count++; // Критический раздел
} finally {
  countLock.unlock(); // непременно снять блокировку
}
```

НА ЗАМЕТКУ. В примерах кода из этого раздела класс **ReentrantLock** применяется для пояснения принципа действия блокировки. Но, как будет показано в следующем разделе, для применения явных блокировок зачастую отсутствует конкретная причина, поскольку имеются неявные блокировки, применяемые по ключевому слову **synchronized**. Тем не менее внутренний механизм блокировки проще понять на примере явных блокировок.

Объект `countLock` блокируется в первом же потоке, где должен быть выполнен метод `lock()`, а затем в этом потоке выполняется критический раздел кода. При попытке вызвать метод `lock()` в другом потоке исполнения для того же самого объекта он будет заблокирован до тех пор, пока не завершится вызов метода `lock()` в первом потоке исполнения. Таким образом, гарантируется, что критический раздел кода будет одновременно выполняться только в одном потоке.

Следует, однако, иметь в виду, что если метод `unlock()` размещается в блоке оператора `finally()`, то блокировка снимается при возникновении любого исключения в критическом разделе кода. В противном случае блокировка будет действовать постоянно, и ни один другой поток исполнения не сможет быть продолжен далее, что, очевидно, очень плохо. Но в рассматриваемом здесь примере исключение вряд ли будет сгенерировано в критическом разделе кода, поскольку в нем выполняется лишь инкрементирование целочисленного значения. Тем не менее для блокировки обычно принято применять блок операторов `try/finally` на тот случай, если впоследствии потребуется ввести дополнительный код.

На первый взгляд, пользоваться блокировками для защиты критических разделов кода нетрудно. Но вся суть кроется в деталях. Как показывает опыт, многие программисты испытывают трудности при написании правильного кода с блокировками. Ведь они могут неверно воспользоваться блокировками или создать условия для *взаимной блокировки*, когда ни один из потоков исполнения не может быть продолжен, поскольку все они ожидают снятия блокировки.

Именно по этой причине прикладным программистам следует прибегать к блокировкам как к последнему средству. Прежде всего следует исключить разделение общих данных, используя вместо них неизменяемые данные или передавая изменяемые данные от одного потока исполнения к другому. Если же разделение общих данных неизбежно, в таком случае следует воспользоваться потокобезопасными структурами данных типа `ConcurrentHashMap` или `LongAdder`. Знание механизма блокировки помогает лучше понять, каким образом реализуются подобные структуры данных, хотя подробности их реализации лучше оставить опытным специалистам.

10.6.2. Ключевое слово synchronized

В предыдущем разделе пояснялось, как пользоваться классом `ReentrantLock` для реализации критического раздела кода. Но пользоваться явной блокировкой вряд ли стоит, поскольку в Java у *каждого объекта* имеется своя *внутренняя блокировка*. Хотя для лучшего понимания внутренних блокировок полезно было рассмотреть сначала явные блокировки.

Ключевое слово synchronized служит для установки внутренней блокировки. Оно применяется в двух формах. Прежде всего, можно заблокировать блок кода следующим образом:

```
synchronized (obj) {
  Критический раздел кода
}
```

Это, по существу, означает следующее:

```
obj.intrinsicLock.lock();
try {
  Критический раздел кода
} finally {
  obj.intrinsicLock.unlock();
}
```

У объекта фактически отсутствует поле с внутренней блокировкой. Приведенный выше код служит лишь для того, чтобы показать, что же происходит, когда применяется ключевое слово synchronized.

Кроме того, метод можно объявить как synchronized. В таком случае тело метода блокируется по ссылке this на параметр получателя. Это означает, что код

```
public synchronized void method() {
  Тело метода
}
```

равнозначен такому коду:

```
public void method() {
  this.intrinsicLock.lock();
  try {
    Тело метода
  } finally {
    this.intrinsicLock.unlock();
  }
}
```

Например, счетчик может быть просто объявлен так, как показано ниже. Благодаря внутренней блокировке экземпляра класса Counter отпадает необходимость в организации явной блокировки.

```
public class Counter {
  private int value;
  public synchronized int increment() {
    value++;
    return value;
  }
}
```

Как видите, применение ключевого слова synchronized делает прикладной код кратким. Разумеется, чтобы понять этот код, нужно знать, что у каждого объекта в Java имеется своя внутренняя блокировка.

НА ЗАМЕТКУ. Блокировки гарантируют не только блокирование, но и доступность данных. Рассмотрим в качестве примера переменную **done**, которая доставила столько хлопот в разделе 10.2.1. Если применить блокировку как для записи, так и для чтения значения переменной **done**, то можно гарантировать, что в коде, вызывающем метод **get()**, будет доступно любое обновление этой переменной через вызов метода **set()**, как показано ниже.

```
public class Flag {
   private boolean done;
   public synchronized void set() { done = true; }
   public synchronized boolean get() { return done; }
}
```

Синхронизированные методы были разработаны по принципу *монитора*, впервые предложенного Пером Бринчем Хансеном (Per Brinch Hansen) и Тони Хоаром (Tony Hoare) в 1970-е годы. По существу, монитор представляет собой класс, в котором все переменные экземпляра являются закрытыми, а все методы защищены закрытой блокировкой.

В языке Java допускается иметь открытые переменные экземпляра и попеременно использовать синхронизированные и несинхронизированные методы. Но дело в том, что внутренняя блокировка открыта для общего доступа.

Многие программисты считают, что это вносит путаницу. Например, в версии Java 1.0 был класс `Hashtable` с синхронизированными методами для изменения хеш-таблицы. Для надежного перебора содержимого такой таблицы можно получить блокировку следующим образом:

```
synchronized (table) {
  for (K key : table.keySet()) ...
}
```

В данном примере параметр `table` обозначает хеш-таблицу и блокировку, которой пользуются ее методы. Такое обозначение служит типичным источником недоразумений (см. упражнение 21 в конце этой главы).

10.6.3. Ожидание по условию

Ниже приведен простой класс `Queue` с методами для ввода и удаления объектов. Синхронизация методов гарантирует, что выполняемые ими операции являются атомарными.

```
public class Queue {
  class Node { Object value; Node next; };
  private Node head;
  private Node tail;

  public synchronized void add(Object newValue) {
    Node n = new Node();
    if (head == null) head = n;
    else tail.next = n;
    tail = n;
    tail.value = newValue;
  }
```

```
  public synchronized Object remove() {
    if (head == null) return null;
    Node n = head;
    head = n.next;
    return n.value;
  }
}
```

А теперь допустим, что метод `remove()` требуется превратить в метод `take()`, который блокируется, если очередь пуста. Проверка на пустоту должна непременно присутствовать в теле синхронизированного метода, как показано ниже. В противном случае она не имеет смысла, поскольку очередь может быть тем временем опустошена.

```
public synchronized Object take() {
  if (head == null) ... // А теперь что?
  Node n = head;
  head = n.next;
  return n.value;
}
```

Но что произойдет, если очередь окажется пустой? Ни в одном другом потоке исполнения нельзя будет ввести элементы до тех пор, пока в текущем потоке исполнения удерживается блокировка. Именно здесь и требуется метод `wait()`. Если оказывается, что выполнение метода `take()` не может быть продолжено, то в нем вызывается метод `wait()`, как показано ниже.

```
public synchronized Object take() throws InterruptedException {
  while (head == null) wait();
  ...
}
```

Текущий поток исполнения теперь деактивизируется, отдавая блокировку. Благодаря этому в другом потоке исполнения появляется возможность вводить элементы в очередь. Это так называемое *ожидание по условию*. Следует иметь в виду, что метод `wait()` определен в классе `Object`. Он имеет отношение к блокировке, связанной с объектом.

Поток исполнения, устанавливающий блокировку с целью ее получения, существенно отличается от потока исполнения, в котором вызывается метод `wait()`. Как только в потоке исполнения вызывается метод `wait()`, он входит в *набор ожидания* для отдельного объекта. Этот поток исполнения не активизируется, как только блокировка становится доступной для него. Вместо этого он остается деактивизированным до тех пор, пока в другом потоке исполнения не будет вызван метод `notifyAll()` для того же самого объекта.

Если в другом потоке исполнения элемент введен в очередь, метод `notifyAll()` должен быть вызван в нем следующим образом:

```
public synchronized void add(Object newValue) {
  ...
  notifyAll();
}
```

В результате вызова метода `notifyAll()` снова активизируются все потоки исполнения в наборе ожидания. Когда потоки исполнения удаляются из набора ожидания, они снова становятся исполняемыми, и в конечном итоге планировщик активизирует их снова. В этот момент они пытаются получить блокировку. И как только одному из них это удастся сделать, он продолжит свое исполнение с того места, где он был остановлен, возвратившись из вызова метода `wait()`.

В этот момент условие должно быть проверено в потоке исполнения еще раз. Ведь нет никакой гарантии, что условие по-прежнему выполняется, поскольку метод `notifyAll()` извещает ожидающие потоки исполнения только о том, что условие *может быть* выполнено к этому моменту. Именно поэтому и стоит проверить условие еще раз. И с этой целью проверка делается в цикле, как показано ниже.

```
while (head == null) wait();
```

ВНИМАНИЕ. Еще один метод, называемый **`notify()`**, разблокирует единственный поток исполнения из набора ожидания. Это более эффективный способ, чем разблокирование всех потоков исполнения сразу, хотя он и более опасный. Ведь если ни в одном другом потоке исполнения метод **`notify()`** больше не вызывается, программа входит в состояние взаимной блокировки.

В потоке исполнения можно вызвать только метод `wait()`, `notifyAll()`, или `notify()` для данного объекта.

НА ЗАМЕТКУ. Методы **`wait()`** и **`notifyAll()`** пригодны для построения структур данных с блокирующими методами. Но пользоваться ими надлежащим образом не так-то просто. Если же от потоков исполнения требуется лишь ожидать до тех пор, пока не будет выполнено заданное условие, то вместо методов **`wait()`** и **`notifyAll()`** лучше воспользоваться одним из классов синхронизации, предоставляемых в параллельной библиотеке, например, классами **`CountdownLatch`** и **`CyclicBarrier`**.

10.7. Потоки исполнения

Эта глава постепенно приближается к своему концу, и поэтому настало время обсудить сами потоки исполнения — примитивы, фактически выполняющие задачи. Как правило, лучше пользоваться исполнителями, автоматически управляющими потоками исполнения. Тем не менее в последующих разделах даются общие сведения о непосредственном обращении с потоками исполнения.

10.7.1. Запуск потока исполнения

Поток исполнения запускается в Java следующим образом:

```
Runnable task = () -> { ... };
Thread thread = new Thread(task);
thread.start();
```

Если же требуется подождать завершения другого потока исполнения, следует вызвать метод join(), как показано ниже. Оба эти метода генерируют проверяемое исключение типа InterruptedException, рассматриваемое в следующем разделе.

```
thread.join(millis);
```

Поток исполнения завершается, когда возврат из его метода run() происходит обычным образом или в связи с генерированием исключения. В последнем случае вызывается *обработчик необрабатываемых исключений* в потоке исполнения. При создании потока исполнения этот обработчик устанавливается в режим обработки необрабатываемых исключений в группе потоков исполнения, и в конечном итоге он становится глобальным обработчиком исключений (см. главу 5). Чтобы сменить обработчик исключений в потоке исполнения, достаточно вызвать метод setUncaughtExceptionHandler().

НА ЗАМЕТКУ. В первоначальной версии Java был определен метод **stop()**, немедленно останавливающий поток исполнения, а также метод **suspend()**, блокирующий поток исполнения до тех пор, пока в другом потоке исполнения не будет вызван метод **resume()**. Эти методы с тех пор не рекомендуются к употреблению.

Метод **stop()** ненадежен по существу. Допустим, что поток исполнения остановлен посредине критического раздела кода, например, ввода элемента в очередь. В таком случае очередь остается в частично обновленном состоянии. Но блокировка, защищающая критический раздел кода, снимается, а испорченная структура данных может быть использована в других потоках исполнения. Прерывать поток исполнения следует лишь в том случае, если требуется остановить его. И тогда прерванный поток исполнения может остановиться лишь в тот момент, когда сделать это безопаснее всего.

Пользоваться методом **suspend()** не столько рискованно, сколько проблематично. Если поток исполнения приостанавливается, не снимая блокировку, то блокируется любой другой поток исполнения, пытающийся получить эту блокировку. Если же возобновить исполнение любого из них, то программа перейдет в состояние взаимной блокировки.

10.7.2. Прерывание потока исполнения

Допустим, что первый же результат по заданному запросу всегда удовлетворителен. Когда же поиск ответа распределяется среди нескольких задач, все остальные задачи требуется отменить, как только ответ будет получен. В языке Java отмена задачи осуществляется *совместно*.

У каждого потока исполнения имеется *прерванное состояние*, указывающее на то, что поток исполнения может быть прерван. Точного определения понятия *прерывание* не существует, но большинство программистов пользуются им для обозначения отмены запроса.

Это состояние может быть проверено средствами интерфейса Runnable. И как правило, это делается в цикле, как показано ниже. Когда поток исполнения прерывается, метод run() просто завершается.

```
Runnable task = () -> {
  while (дополнительно выполняемые операции) {
    if (Thread.currentThread().isInterrupted()) return;
```

```
      выполнить дополнительные операции
  }
};
```

НА ЗАМЕТКУ. Имеется также статический метод **`Thread.interrupted()`**, получающий прерванное состояние текущего потока исполнения, сбрасывающий его и возвращающий прежнее состояние.

Иногда поток исполнения становится временно неактивным. Такое может произойти в том случае, если поток исполнения ожидает значение, которое должно быть вычислено в другом потоке исполнения, или ввод-вывод данных либо переходит в состояние бездействия, чтобы предоставить другим потокам возможность для исполнения.

Если же поток исполнения прерывается в состоянии ожидания или бездействия, он сразу же активизируется снова, но на этот раз прерванное состояние не устанавливается. Вместо этого генерируется исключение типа `InterruptedException`. Это проверяемое исключение, и поэтому оно должно быть перехвачено в теле метода `run()` из интерфейса `Runnable`. Обычной реакцией на это исключение является завершение метода `run()`, как показано ниже.

```
Runnable task = () -> {
  try {
    while (дополнительно выполняемые операции) {
      выполнить дополнительные операции
      Thread.sleep(millis);
    }
  }
  catch (InterruptedException ex) {
    // Ничего не делать
  }
};
```

Если исключение типа `InterruptedException` перехватывается именно таким образом, то проверять прерванное состояние не нужно. А если поток исполнения был прерван за пределами вызова метода `Thread.sleep()`, то прерванное состояние устанавливается, а метод `Thread.sleep()` сгенерирует исключение типа `InterruptedException`, как только он будет вызван.

СОВЕТ. Исключение типа **`InterruptedException`** может показаться досадным, но его недостаточно перехватить и скрыть, когда вызывается такой метод, как **`sleep()`**. Если нельзя больше ничего сделать, то можно хотя бы установить прерванное состояние следующим образом:

```
try {
  Thread.sleep(millis);
} catch (InterruptedException ex) {
  Thread.currentThread().interrupt();
}
```

А еще лучше передать исключение компетентному обработчику таким образом:

```
public void mySubTask() throws InterruptedException {
```

```
        Thread.sleep(millis);
        ...
    }
```

10.7.3. Локальные переменные в потоках исполнения

Иногда разделения общих данных среди потоков исполнения можно избежать, предоставив каждому потоку отдельный экземпляр с помощью вспомогательного класса `ThreadLocal`. Например, экземпляры класса `NumberFormat` не являются потокобезопасными. Допустим, что имеется следующая статическая переменная:

```
public static final NumberFormat currencyFormat =
      NumberFormat.getCurrencyInstance();
```

Если в двух потоках исполняется операция вроде следующей:

```
String amountDue = currencyFormat.format(total);
```

то конечный результат может превратиться в "мусор", поскольку внутренние структуры данных, используемые экземпляром класса `NumberFormat`, могут быть нарушены при параллельном доступе. В качестве выхода из этого положения можно было бы воспользоваться синхронизацией, но это было бы затратно. С другой стороны, локальный объект типа `NumberFormat` можно было бы строить всякий раз, когда бы в нем возникала потребность, но и это было бы расточительно.

Чтобы построить по одному экземпляру на каждый поток исполнения, можно воспользоваться следующим фрагментом кода:

```
public static final ThreadLocal<NumberFormat> currencyFormat =
    ThreadLocal.withInitial(() -> NumberFormat.getCurrencyInstance());
```

А для того чтобы получить доступ к конкретному средству форматирования, достаточно сделать следующий вызов:

```
String amountDue = currencyFormat.get().format(total);
```

Когда метод `get()` вызывается в отдельном потоке исполнения в первый раз, в конструкторе вызывается лямбда-выражение, чтобы получить экземпляр для данного потока исполнения. А далее метод `get()` возвращает экземпляр, относящийся к текущему потоку исполнения.

10.7.4. Различные свойства потоков исполнения

В классе `Thread` доступен целый ряд свойств потоков исполнения, но большинство из этих свойств полезны скорее учащимся, которые готовятся к экзаменам на присвоение квалификации, чем прикладным программистам. В этом разделе дается краткий обзор подобных свойств.

Потоки исполнения можно собирать в группы, и в прикладном программном интерфейсе API имеются методы для управления группами потоков исполнения,

включая прерывание всех потоков в группе. В настоящее время более предпочтительным механизмом управления группами задач считаются исполнители.

Для потоков исполнения можно задать *свойства*, чтобы потоки с высоким приоритетом исполнялись прежде потоков с низким приоритетом. Можно надеяться, что свойства принимаются во внимание виртуальной машиной и главной платформой, но подробности сильно зависят от конкретной платформы. Следовательно, пользоваться свойствами ненадежно и обычно не рекомендуется.

У потоков исполнения имеются *состояния*, по которым можно определить, является ли поток новым, исполняемым, блокированным по вводу-выводу, ожидающим или завершенным. Прикладным программистам, пользующимся потоками исполнения, редко приходится запрашивать их состояние.

Когда поток исполнения прерывается из-за необрабатываемого исключения, последнее передается *обработчику необрабатываемых исключений* в этом потоке исполнения. По умолчанию трассировка его стека направляется в стандартный поток вывода ошибок `System.err`, но имеется возможность установить свой обработчик (см. главу 5).

Потоковый демон — это поток исполнения, единственное назначение которого служить другим потокам. Он полезен для тех потоков исполнения, которые посылают такты времени, отсчитываемые таймером, или очищают устаревшие записи в кеше. Когда остаются только потоковые демоны, виртуальная машина завершает работу. Чтобы создать потоковый демон, следует вызвать метод `thread.setDaemon(true)`, прежде чем запускать поток исполнения.

10.8. Асинхронные вычисления

До сих пор рассматривался подход к параллельным вычислениям, состоявший в разбиении задачи на части и ожидании до тех пор, пока все ее части будут завершены. Но ожидание не всегда целесообразно. В последующих разделах поясняется, как реализовать вычисления без ожидания, называемые иначе *асинхронными*.

10.8.1. Длительные задачи в обратных вызовах пользовательского интерфейса

Одной из причин применения потоков исполнения служит ускорение реагирования прикладных программ на внешние воздействия. Это особенно важно для прикладных программ с пользовательским интерфейсом. Если в прикладной программе требуется выполнить задачу, отнимающую время, этого нельзя делать в потоке исполнения пользовательского интерфейса, иначе он перестанет реагировать на действия пользователя. Вместо этого следует запустить другой рабочий поток исполнения.

Так, если требуется прочитать веб-страницу, когда пользователь щелкает на экранной кнопке, этого не стоит делать следующим образом:

```
Button read = new Button("Read");
read.setOnAction(event -> { // Неудачно, т.к. действие происходит
              // в потоке исполнения пользовательского интерфейса
  Scanner in = new Scanner(url.openStream());
  while (in.hasNextLine()) {
```

```
    String line = in.nextLine();
    ...
  }
});
```

Вместо этого данную операцию лучше выполнить в отдельном потоке таким образом:

```
read.setOnAction(event -> { // Удачно, т.к. длительное действие
                            // выполняется в отдельном потоке
  Runnable task = () -> {
    Scanner in = new Scanner(url.openStream());
    while (in.hasNextLine()) {
      String line = in.nextLine();
      ...
    }
  }
  new Thread(task).start();
});
```

Следует, однако, очень внимательно отнестись к тому, что делается в рабочем потоке исполнения. Пользовательские интерфейсы, построенные на основе библиотек JavaFX и Swing или на платформе Android, не являются потокобезопасными. Если манипулировать элементами пользовательского интерфейса из нескольких потоков исполнения, то эти элементы могут быть испорчены. В действительности этот факт проверяется в JavaFX и Android, и генерируется исключение, если попытаться получить доступ к пользовательскому интерфейсу из другого потока исполнения, а не из потока исполнения пользовательского интерфейса.

Таким образом, любые обновления пользовательского интерфейса в потоке его исполнения нужно планировать. В каждой библиотеке пользовательского интерфейса предоставляется свой механизм планирования задачи типа `Runnable` для исполнения в потоке пользовательского интерфейса. Например, в библиотеке JavaFX для этой цели можно воспользоваться следующим вызовом:

```
Platform.runLater(() ->
    message.appendText(line + "\n"));
```

НА ЗАМЕТКУ. Реализовать длительные операции таким образом, чтобы своевременно реагировать на действия пользователя по ходу выполнения этих операций, не так-то просто. Поэтому библиотеки пользовательского интерфейса обычно предоставляют какой-нибудь вспомогательный класс, например, **`SwingWorker`** в библиотеке Swing или **`AsyncTask`** на платформе Android, реализующий подробности данного процесса. А программисту достаточно указать действия для длительной задачи, которая выполняется в отдельном потоке, а также постепенность обновлений и окончательную диспозицию (что обычно делается в потоке исполнения пользовательского интерфейса).

В классе **`Task`** из библиотеки JavaFX принят несколько иной подход к постепенности обновлений. В этом классе предоставляются методы для обновления свойств задачи (сообщения, доли выполнения в процентах и результирующего значения) в длительном потоке исполнения. Эти свойства привязываются к элементам пользовательского интерфейса, которые затем обновляются в потоке исполнения пользовательского интерфейса.

10.8.2. Завершаемые будущие действия

Традиционный подход к обращению с неблокирующими вызовами заключается в том, чтобы пользоваться обработчиками событий, программно регистрируемых для действий, которые должны произойти по завершении задачи. Разумеется, если следующее действие также является асинхронным, то и следующее после него действие должно происходить в другом обработчике событий. Несмотря на то что программист мыслит следующими категориями: сначала сделать шаг 1, затем шаг 2 и далее шаг 3, логика программы становится распределенной среди разных обработчиков. А добавление обработки ошибок только усложняет дело. Допустим, что на шаге 2 пользователь входит в систему. Этот шаг, возможно, придется повторить, поскольку пользователь может ввести свои учетные данные с опечатками. Реализовать такой поток управления в ряде обработчиков событий непросто, а еще труднее другим понять, как он был реализован.

Совсем иной поход предоставляется в классе `CompletableFuture`. В отличие от обработчиков событий, завершаемые будущие действия могут быть *составлены*. Допустим, что с веб-страницы требуется извлечь все ссылки, чтобы построить поисковый робот. Допустим также, что для этой цели имеется следующий метод, получающий текст с веб-страницы, как только он становится доступным:

```
public void CompletableFuture<String> readPage(URL url)
```

Если следующий метод:

```
public static List<URL> getLinks(String page)
```

получает URL на HTML-странице, то его вызов можно запланировать на момент, когда страница доступна, следующим образом:

```
CompletableFuture<String> contents = readPage(url);
CompletableFuture<List<URL>> links = contents.thenApply(Parser::getLinks);
```

Метод `thenApply()` вообще не блокируется. Он возвращает другое будущее действие. По завершении первого будущего действия его результат передается методу `getLinks()`, а значение, возвращаемое этим методом, становится окончательным результатом.

Применяя завершаемые будущие действия, достаточно указать, что и в каком порядке требуется сделать. Безусловно, все это происходит не сразу, но самое главное, что весь код оказывается в одном месте.

Принципиально класс `CompletableFuture` является простым прикладным программным интерфейсом API, но для составления завершаемых будущих действий имеется немало вариантов его методов. Рассмотрим сначала те из них, которые обращаются с единственным будущим действием. Для каждого метода, перечисленного в табл. 10.2, имеются еще два варианта типа `Async`, которые в этой таблице не представлены. В одном из этих вариантов используется общий объект типа `ForkJoinPool`, а в другом имеется параметр типа `Executor`. Кроме того, в табл. 10.2 употребляется следующее сокращенное обозначение громоздких функциональных интерфейсов:

T -> U вместо Function<? super T, U>. Обозначения **T** и **U**, разумеется, не имеют никакого отношения к конкретным типам данных в Java.

Метод thenApply() уже был представлен выше. Так, в результате следующих вызовов:

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

возвращается будущее действие, применяющее функцию f к результату будущего действия future, как только он станет доступным. А в результате второго вызова функция f выполняется еще в одном потоке.

Вместо функции T -> U метод thenCompose() принимает функцию T -> CompletableFuture<U>. На первый взгляд это кажется довольно абстрактным, но может быть вполне естественным. Рассмотрим действие чтения веб-страницы по заданному URL. Вместо того чтобы вызывать следующий метод:

```
public String blockingReadPage(URL url)
```

изящнее возвратить из метода будущее действие следующим образом:

```
public CompletableFuture<String> readPage(URL url)
```

А теперь допустим, что имеется еще один, приведенный ниже метод, получающий URL из вводимых пользователем данных, возможно, в диалоговом окне, где ответ не появляется до тех пор, пока пользователь не щелкнет на экранной кнопке OK. И это считается событием в будущем действии.

```
public CompletableFuture<URL> getURLInput(String prompt)
```

В данном случае имеются две функции: T -> CompletableFuture<U> и U -> CompletableFuture<V>. Очевидно, что они составляют функцию T -> CompletableFuture<V>, если вызывается вторая функция, когда завершается первая. Именно это и делается в методе thenCompose().

Третий метод из табл. 10.2 сосредоточен на отказе — другом, игнорировавшемся до сих пор аспекте. Когда генерируется исключение типа CompletableFuture, оно перехватывается и заключается в оболочку непроверяемого исключения типа ExecutionException при вызове метода get(). Но возможно, метод get() не будет вызван вообще. Чтобы обработать это исключение, следует вызвать метод handle(). Предоставляемая ему функция вызывается с результатом (а в его отсутствие — с пустым значением null) и исключением (а в его отсутствие — с пустым значением null), что имеет смысл в данном случае. Остальные методы возвращают результат типа void и, как правило, применяются в конце конвейера обработки.

Таблица 10.2. Методы ввода будущего действия в объект типа CompletableFuture<T>

Метод	Параметр	Описание
thenApply()	**T -> U**	Применить функцию к результату
thenCompose()	**T -> CompletableFuture<U>**	Вызвать функцию для результата и выполнить возвращаемое будущее действие

Окончание табл. 10.2

Метод	Параметр	Описание
`handle()`	`(T, Throwable) -> U`	Обработать результат или ошибку
`thenAccept()`	`T -> void`	Аналогично методу **`thenApply()`**, но с результатом типа **`void`**
`whenComplete()`	`(T, Throwable) -> void`	Аналогично методу **`handle()`**, но с результатом типа **`void`**
`thenRun()`	`Runnable`	Выполнить задачу типа **`Runnable`** с результатом типа **`void`**

А теперь рассмотрим методы, объединяющие несколько будущих действий (табл. 10.3). Три первых метода выполняют действия типа `CompletableFuture<T>` и `CompletableFuture<U>` параллельно и объединяют полученные результаты.

Следующие три метода выполняют два действия типа `CompletableFuture<T>` параллельно. Как только одно из них завершается, передается его результат, а результат другого действия игнорируется.

И наконец, статические методы `allOf()` и `anyOf()` принимают переменное количество завершаемых будущих действий и получают завершаемое действие типа `CompletableFuture<Void>`, которое завершается, когда завершаются все они или одно из них. В таком случае результаты не распространяются.

НА ЗАМЕТКУ. Формально говоря, методы, рассматриваемые в этом разделе, принимают параметры типа **`CompletionStage`**, а не типа **`CompletableFuture`**. Интерфейс **`CompletionStage`** состоит из почти сорока абстрактных методов, реализуемых в классе **`CompletableFuture`**. Этот интерфейс предоставляется для того, чтобы его можно было реализовать в сторонних каркасах.

Таблица 10.3. Методы объединения нескольких объектов составления будущих действий

Метод	Параметры	Описание
`thenCombine()`	`CompletableFuture<U>, (T, U) -> V`	Выполнить оба действия и объединить полученные результаты с помощью заданной функции
`thenAcceptBoth()`	`CompletableFuture<U>, (T, U) -> void`	Аналогично методу **`thenCombine()`**, но с результатом типа **`void`**
`runAfterBoth()`	`CompletableFuture<?>, Runnable`	Выполнить задачу типа **`Runnable`** по завершении обоих действий
`applyToEither()`	`CompletableFuture<T>, T -> V`	Если доступен результат выполнения одного или другого действия, передать его заданной функции
`acceptEither()`	`CompletableFuture<T>, T -> void`	Аналогично методу **`applyToEither()`**, но с результатом типа **`void`**
`runAfterEither()`	`CompletableFuture<?>, Runnable`	Выполнить задачу типа **`Runnable`** по завершении одного или другого действия

Окончание табл. 10.3

Метод	Параметры	Описание
`static allOf()`	`CompletableFuture<?>...`	Завершить с результатом типа `void` по окончании всех заданных будущих действий
`static anyOf()`	`CompletableFuture<?>...`	Завершить с результатом типа `void` по окончании любого из заданных будущих действий

10.9. Процессы

До сих пор было показано, как выполнять написанный на Java код в отдельных потоках исполнения из одной и той же программы. Но иногда требуется выполнить и другую программу. Для этой цели служат классы `ProcessBuilder` и `Process`. В частности, класс `Process` служит для выполнения команды в отдельном процессе операционной системы и позволяет взаимодействовать с ее стандартными потоками ввода-вывода данных и ошибок. А класс `ProcessBuilder` дает возможность настраивать соответствующим образом объект типа `Process`.

НА ЗАМЕТКУ. Классом **`ProcessBuilder`** удобнее пользоваться для замены вызовов **`Runtime.exec()`**.

10.9.1. Построение процесса

Начать построение процесса следует с указания команды, которую требуется выполнить. Для этого достаточно предоставить конструктору класса `ProcessBuilder` список типа `List<String>` или просто символьные строки, составляющие команду, как показано ниже.

```
ProcessBuilder builder = new ProcessBuilder("gcc", "myapp.c");
```

ВНИМАНИЕ. Первая символьная строка должна содержать исполняемую, а не встроенную в командную оболочку команду. Например, чтобы выполнить команду **`dir`** в Windows, нужно построить процесс с символьными строками **`"cmd.exe"`**, **`"/C"`** и **`"dir"`**.

У каждого процесса имеется свой *рабочий каталог*, применяемый для разрешения относительных путей к каталогам. По умолчанию рабочий каталог процесса такой же, как и у виртуальной машины, и, как правило, это каталог, из которого выполняется утилита **`java`** с запускаемой программой. С помощью метода `directory()` этот каталог можно изменить следующим образом:

```
builder = builder.directory(path.toFile());
```

НА ЗАМЕТКУ. Каждый из методов настройки объекта типа **ProcessBuilder** возвращает результат, и поэтому команды можно связать в цепочку. В конечном итоге получается вызов, аналогичный следующему:

```
Process p = new ProcessBuilder(команда).directory(файл)....start();
```

Далее требуется указать, что делать со стандартными потоками ввода-вывода данных и ошибок в процессе. По умолчанию каждый из них доступен как конвейер следующим образом:

```
OutputStream processIn = p.getOutputStream();
InputStream processOut = p.getInputStream();
InputStream processErr = p.getErrorStream();
```

Следует, однако, иметь в виду, что поток ввода в процессе служит потоком вывода в виртуальной машине JVM! Все, что направляется в этот поток вывода, становится вводом в процесс. С другой стороны, можно ввести все, что процесс направляет в потоки вывода данных и ошибок. В этом случае они служат потоками ввода.

Для нового процесса можно указать те же самые потоки ввода-вывода данных и ошибок, что и для виртуальной машины JVM. Если пользователь выполняет виртуальную машину JVM на консоли, то любые вводимые им данные направляются процессу, а результат выполнения процесса выводится на консоль. Чтобы настроить подобным образом все три потока ввода-вывода, достаточно сделать следующий вызов:

```
builder.redirectIO()
```

А если требуется только наследовать некоторые из потоков ввода-вывода, то методу `redirectInput()`, `redirectOutput()` или `redirectError()` достаточно передать в качестве параметра значение `ProcessBuilder.Redirect.INHERIT`, как показано в следующем примере кода:

```
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

Потоки ввода-вывода можно переадресовывать из процесса в файлы, предоставляя объекты типа `File` следующим образом:

```
builder.redirectInput(inputFile)
    .redirectOutput(outputFile)
    .redirectError(errorFile)
```

Файлы для вывода данных и ошибок создаются и укорачиваются при запуске процесса. Чтобы присоединить уже имеющиеся файлы, можно воспользоваться следующим вызовом:

```
builder.redirectOutput(ProcessBuilder.Redirect.appendTo(outputFile));
```

Нередко потоки вывода данных и ошибок удобно объединить вместе, чтобы наблюдать выводимые результаты и ошибки в той последовательности, в какой они формируются в процессе. Для этой цели служит следующий вызов:

```
builder.redirectErrorStream(true)
```

В таком случае можно больше не вызывать метод `redirectError()` для объекта типа `ProcessBuilder` или метод `getErrorStream()` для объекта типа `Process`.

И наконец, может возникнуть потребность внести изменения в переменные окружения процесса. Но для этой цели уже не годится синтаксис построения процесса по цепочке. Ведь нужно сначала получить переменную окружения построителя процесса, которая инициализируется переменными экземпляра процесса, выполняющего виртуальную машину JVM, а затем ввести или удалить соответствующие записи, как показано ниже.

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

10.9.2. Выполнение процесса

После настройки построителя процесса вызывается его метод `start()` для запуска процесса. Если потоки ввода-вывода ошибок настроены как конвейеры, то теперь можно выполнять запись в поток ввода и чтение из потоков ввода данных и ошибок, как показано в следующем примере кода:

```
Process p = new ProcessBuilder("/bin/ls", "-l")
    .directory(Paths.get("/tmp").toFile())
    .start();
try (Scanner in = new Scanner(p.getInputStream())) {
  while (in.hasNextLine())
    System.out.println(in.nextLine());
}
```

ВНИМАНИЕ. Для буфера потоков ввода-вывода отдельного процесса выделяется ограниченное пространство. Поэтому поток ввода не следует переполнять данными, а из потока вывода нужно своевременно читать данные. Если же имеется немало данных для ввода-вывода, то поставлять и потреблять их, возможно, придется в отдельных потоках исполнения.

Чтобы ожидать завершения процесса, достаточно сделать следующий вызов:

```
int result = p.waitFor();
```

А для того чтобы не ожидать завершения процесса бесконечно, достаточно организовать код следующим образом:

```
long delay = ...;
if (p.waitfor(delay, TimeUnit.SECONDS)) {
  int result = p.exitValue();
  ...
} else {
  p.destroyForcibly();
}
```

В первом случае в результате вызова метода `waitFor()` возвращается значение выхода из процесса (по умолчанию нулевое значение обозначает успешное завершение

процесса, а ненулевое значение — код ошибки). А во втором случае в результате вызова метода `waitFor()` возвращается логическое значение `true`, если не истекло время ожидания процесса. В таком случае значение выхода из процесса придется извлечь, вызвав метод `exitValue()`.

Вместо того чтобы ожидать процесс, можно просто предоставить ему возможность выполняться и периодически вызывать метод `isAlive()`, чтобы проверить, действует ли он по-прежнему. Чтобы удалить процесс, достаточно вызвать метод `destroy()` или `destroyForcibly()`. Отличия в этих методах зависят от конкретной платформы. Так, в Unix первый из них удаляет процесс с параметром `SIGTERM`, а второй — с параметром `SIGKILL`.

Упражнения

1. Используя параллельные потоки данных, найдите в каталоге все файлы, содержащие заданное слово. Как найти только первый файл? Действительно ли поиск файлов осуществляется параллельно?
2. Насколько большим должен быть массив, чтобы метод `Arrays.parallelSort()` выполнялся быстрее, чем метод `Arrays.sort()` на вашем компьютере?
3. Реализуйте метод, возвращающий задачу для чтения всех слов из файла с целью найти в нем заданное слово. Если задача прерывается, она должна быть завершена немедленно с выдачей отладочного сообщения. Запланируйте выполнение этой задачи для каждого файла в каталоге. Как только одна из задач завершится успешно, все остальные задачи должны быть немедленно прерваны.
4. В разделе 10.3.2 не была рассмотрена еще одна параллельная операция над массивами. Она реализуется в методе `parallelPrefix()` и заменяет каждый элемент массива накоплением префикса для заданной ассоциативной операции. В качестве примера рассмотрим массив `[1, 2, 3, 4, ...]` и операцию `×`. После выполнения вызова `Arrays.parallelPrefix(values, (x, y) -> x * y)` массив содержит следующие элементы:

   ```
   [1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, ...]
   ```

   Как ни странно, подобное вычисление можно распараллелить. Сначала нужно соединить смежные элементы, как показано ниже.

   ```
   [1, 1 × 2, 3, 3 × 4, 5, 5 × 6, 7, 7 × 8]
   ```

   Значения, выделенные обычным шрифтом, не затрагиваются данной операцией. Очевидно, что ее можно выполнить параллельно на отдельных участках массива, а затем обновить выделенные выше полужирным элементы, перемножив их с элементами, находящимися на одну или две позиции раньше, как показано ниже.

   ```
   [1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, 5, 5 × 6, 5 × 6 × 7, 5 × 6 × 7 × 8]
   ```

И эту операцию можно выполнить параллельно. После `log(n)` шагов процесс будет завершен. Это более выгодный способ, чем простое линейное вычисление при наличии достаточного количества процессоров.

Воспользуйтесь в данном упражнении методом `parallelPrefix()`, чтобы распараллелить вычисление чисел Фибоначчи, используя тот факт, что n-е число Фибоначчи является левым верхним коэффициентом F^n, где $F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Создайте массив, заполнив его матрицами `2×2`. Определите класс `Matrix` с методом умножения. Воспользуйтесь методом `parallelSetAll()`, чтобы создать массив матриц, а методом `parallelPrefix()`, чтобы перемножить их.

5. Напишите прикладную программу, где в нескольких потоках исполнения читаются все слова из совокупности файлов. Воспользуйтесь параллельным отображением типа `ConcurrentHashMap<String, Set<File>>` для отслеживания файлов, в которых встречается каждое слово, а методом `merge()` — для обновления данного отображения.
6. Повторите предыдущее упражнение, но на этот раз воспользуйтесь методом `computeIfAbsent()`. В чем преимущество такого подхода?
7. Найдите в отображении типа `ConcurrentHashMap<String, Long>` ключ с максимальным значением, произвольно отбрасывая лишнее. *Подсказка:* воспользуйтесь методом `reduceEntries()`.
8. Сформируйте 1000 потоков исполнения, в которых счетчик инкрементируется 100000 раз. Сравните производительность при использовании классов `AtomicLong` и `LongAdder`.
9. Воспользуйтесь классом `LongAccumulator` для вычисления максимального и минимального накапливаемых элементов.
10. Воспользуйтесь блокирующей очередью для обработки файлов в каталоге. В одном потоке исполнения организуйте обход дерева каталога и ввод файлов в очередь, а в нескольких потоках исполнения — удаление файлов и поиск в каждом из них заданного ключевого слова с выводом любых совпадений. Как только задача поставщика будет завершена, в очередь должен быть введен фиктивный файл.
11. Повторите предыдущее упражнение, но на этот раз сформируйте в задаче потребителя отображение слов и частоты, с которой они вводятся во вторую очередь. В последнем потоке исполнения полученные словари должны быть объединены, а затем выведены десять наиболее употребительных слов. Почему для этой цели не потребуется отображение типа `ConcurrentHashMap`?
12. Повторите предыдущее упражнение, создав объект типа `Callable<Map<String, Integer>>` для каждого файла и воспользовавшись подходящей службой исполнителя. Объедините полученные результаты, как только все они будут доступны. Почему для этой цели не потребуется отображение типа `ConcurrentHashMap`?

13. Повторите предыдущее упражнение, воспользовавшись на этот раз интерфейсом `ExecutorCompletionService` и объединив полученные результаты, как только все они будут доступны.
14. Повторите предыдущее упражнение, воспользовавшись глобальным отображением типа `ConcurrentHashMap` для накопления частоты, с которой встречается каждое слово.
15. Повторите предыдущее упражнение, воспользовавшись параллельными потоками данных. Ни одна из потоковых операций не должна иметь никаких побочных эффектов.
16. Напишите программу для обхода дерева каталогов с формированием отдельного потока исполнения для каждого файла. Подсчитайте в потоках исполнения количество слов в файлах и, не прибегая к блокировкам, обновите общий счетчик, который объявляется следующим образом:

    ```
    public static long count = 0;
    ```

    Выполните эту программу неоднократно. Что при этом происходит и почему?
17. Усовершенствуйте программу из предыдущего упражнения, используя блокировки.
18. Усовершенствуйте программу из предыдущего упражнения, используя класс `LongAdder`.
19. Рассмотрите следующую реализацию стека:

    ```
    public class Stack {
      class Node { Object value; Node next; };
      private Node top;

      public void push(Object newValue) {
        Node n = new Node();
        n.value = newValue;
        n.next = top;
        top = n;
      }

      public Object pop() {
        if (top == null) return null;
        Node n = top;
        top = n.next;
        return n.value;
      }
    }
    ```

    Опишите два разных пути, приводящих к тому, что данная структура данных может содержать неверные элементы.
20. Рассмотрите следующую реализацию очереди:

    ```
    public class Queue {
      class Node { Object value; Node next; };
      private Node head;
      private Node tail;
    ```

```
  public void add(Object newValue) {
    Node n = new Node();
    if (head == null) head = n;
    else tail.next = n;
    tail = n;
    tail.value = newValue;
  }

  public Object remove() {
    if (head == null) return null;
    Node n = head;
    head = n.next;
    return n.value;
  }
}
```

21. Опишите два разных пути, приводящих к тому, что данная структура данных может содержать неверные элементы.

22. Найдите ошибку в следующем фрагменте кода:

```
public class Stack {
  private Object myLock = "LOCK";

  public void push(Object newValue) {
    synchronized (myLock) {
    ...
    }
  }
  ...
}
```

23. Найдите ошибку в следующем фрагменте кода:

```
public class Stack {
  public void push(Object newValue) {
    synchronized (new ReentrantLock()) {
      ...
    }
  }
  ...
}
```

24. Найдите ошибку в следующем фрагменте кода:

```
public class Stack {
  private Object[] values = new Object[10];
  private int size;

  public void push(Object newValue) {
    synchronized (values) {
      if (size == values.length)
        values = Arrays.copyOf(values, 2 * size);
      values[size] = newValue;
      size++;
    }
  }
  ...
}
```

25. Напишите программу, запрашивающую у пользователя URL, читающую веб-страницу по этому URL и выводящую на экран все ссылки на ней. Воспользуйтесь для каждой из этих стадий классом `CompletableFuture`. Только не вызывайте метод `get()`. Чтобы не допустить преждевременного прекращения работы данной программы, сделайте следующий вызов:

```
ForkJoinPool.commonPool().awaitQuiescence(10, TimeUnit.SECONDS);
```

26. Напишите следующий метод:

```
public static <T> CompletableFuture<T> repeat(
    Supplier<T> action, Predicate<T> until)
```

Этот метод должен асинхронно повторять заданное действие до тех пор, пока не будет получено значение, принимаемое функцией `until()`, которая также должна выполняться асинхронно. Проверьте этот метод с помощью одной функции, вводящей объект типа `java.net.PasswordAuthentication` с консоли, и другой функции, имитирующей проверку достоверности, ожидая в течение секунды и затем проверяя пароль `"secret"`. *Подсказка:* воспользуйтесь рекурсией.

Глава 11

Аннотации

В этой главе...

Аннотации являются дескрипторами, вставляемыми в исходный код с целью обработать их какими-нибудь инструментальными средствами. Эти средства могут действовать на уровне исходного кода или обрабатывать файлы классов, в которых компилятор разместил аннотации.

Аннотации не изменяют порядок компиляции программы. Компилятор Java генерирует те же самые инструкции для виртуальной машины как при наличии аннотаций, так и в их отсутствие. Чтобы извлечь выгоду из аннотаций, нужно выбрать подходящее инструментальное средство их обработки и пользоваться теми аннотациями, которые оно понимает, прежде чем применять его в своем коде.

Аннотации находят широкое применение. Например, в инструментальном средстве блочного тестирования JUnit они применяются для пометки методов, выполняющих тесты, а также для указания порядка их выполнения. А в архитектуре Java Persistence Architecture аннотации служат для определения взаимных преобразований классов и таблиц базы данных, чтобы объекты могли сохраняться автоматически, не вынуждая разработчиков писать для этой цели запросы SQL.

В этой главе рассматриваются особенности синтаксиса аннотаций, определения собственных аннотаций и написания процессоров аннотаций, работающих на уровне исходного кода во время выполнения.

Основные положения этой главы приведены ниже.

1. Аннотировать можно объявления аналогично употреблению таких модификаторов доступа, как `public` или `static`.
2. Аннотировать можно также типы данных, появляющиеся в объявлениях, приведение и проверки типов `instanceof` или ссылки на методы.
3. Аннотации начинаются со знака `@` и могут содержать пары "ключ–значение", называемые элементами аннотаций.
4. Значениями аннотаций должны быть константы времени компиляции, т.е. статические константы: примитивных типов, перечислимого типа, литералы типа `Class`, другие аннотации или их массивы.
5. Элемент кода может быть снабжен повторяющимися или разнотипными аннотациями.
6. Чтобы определить аннотацию, достаточно указать ее интерфейс, методы которого соответствуют элементам аннотации.
7. В библиотеке Java определяется более десятка аннотаций, которые широко применяются в версии Java Enterprise Edition.
8. Для обработки аннотаций в выполняющейся программе на Java можно воспользоваться рефлексией, запрашивая получаемые с ее помощью элементы кода для аннотаций.
9. Процессоры аннотаций служат для обработки исходных файлов во время компиляции, используя прикладной программный интерфейс API модели языка для обнаружения аннотированных элементов кода.

11.1. Применение аннотаций

Ниже приведен пример простой аннотации.

```
public class CacheTest {
  ...
  @Test public void checkRandomInsertions()
}
```

В данном примере аннотация `@Test` служит для аннотирования метода `checkRandomInsertions()`. В языке Java аннотация применяется аналогично модификатору доступа вроде `public` или `static`. Имя каждой аннотации предваряется знаком `@`.

Сама аннотация `@Test` ничего не делает. Для того чтобы она принесла какую-то пользу, требуется специальное инструментальное средство. Например, в инструментальном средстве тестирования JUnit 4, свободно доступном по адресу `http://junit.org`, вызовы всех методов помечаются аннотацией `@Test` при тестировании их класса. А другое инструментальное средство может удалить все тестовые методы из файла класса, чтобы они не поставлялись вместе с программой после ее тестирования.

11.1.1. Элементы аннотаций

У аннотаций могут быть пары "ключ–значение", называемые *элементами*, как показано ниже.

```
@Test(timeout=10000)
```

Имена и типы допустимых элементов определяются каждой аннотацией (см. далее в разделе 11.2). Элементы аннотаций могут быть обработаны инструментальными средствами, читающими аннотации.

К числу элементов аннотаций относятся следующие.

- Значение примитивного типа.
- Объект типа `String`.
- Объект типа `Class`.
- Экземпляр перечислимого типа.
- Собственно аннотация.
- Массив любых из перечисленных выше элементов, но не массив массивов.

В качестве примера ниже приведена аннотация, состоящая из разнотипных элементов.

```
@BugReport(showStopper=true,
  assignedTo="Harry",
  testCase=CacheTest.class,
  status=BugReport.Status.CONFIRMED)
```

ВНИМАНИЕ. Элемент аннотации вообще не может иметь пустое значение `null`.

Элементы аннотаций могут иметь значения, устанавливаемые по умолчанию. Например, значение по умолчанию элемента `timeout` аннотации `@Test` равно `0L`. Следовательно, аннотация `@Test` равнозначна аннотации `@Test(timeout=0L)`.

Если элемент аннотации называется `value` и является указанным в ней единственным элементом, то присваивание `value=` можно опустить. Например, аннотация `@SuppressWarnings("unchecked")` равнозначна аннотации `@SuppressWarnings (value="unchecked")`.

Если в качестве значения аннотации присваивается массив, то его элементы заключаются в фигурные скобки, как показано ниже.

```
@BugReport(reportedBy={"Harry", "Fred"})
```

Фигурные скобки можно опустить, если массив состоит из единственного элемента, как в следующей строке кода:

```
@BugReport(reportedBy="Harry") // То же, что и {"Harry"}
```

В качестве элемента одной аннотации может служить другая аннотация, как показано ниже.

```
@BugReport(ref=@Reference(id=11235811), ...)
```

ВНИМАНИЕ. Аннотации вычисляются компилятором, и поэтому все значения их элементов должны быть константами времени компиляции, т.е. статическими константами.

11.1.2. Многие и повторяющиеся аннотации

Элемент кода может быть помечен несколькими аннотациями, как показано ниже.

```
@Test
@BugReport(showStopper=true, reportedBy="Joe")
public void checkRandomInsertions()
```

Если автор аннотации объявил ее как повторяющуюся, то одну и ту же аннотацию можно повторить неоднократно следующим образом:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```

11.1.3. Объявление аннотаций

До сих пор были представлены примеры аннотаций, применявшихся в объявлениях методов. Аннотации могут встречаться и во многих других местах прикладного кода. Эти места можно разделить на две категории: *объявления* и *места употребления типов*. Аннотации могут появляться в объявлениях следующих элементов кода.

- Классы (включая и перечисления) и интерфейсы (в том числе и интерфейсы аннотаций).

- Методы.
- Конструкторы.
- Переменные экземпляра (включая и константы перечислимого типа).
- Локальные переменные (в том числе и те, что объявлены в цикле `for` и операторах `try` с ресурсами).
- Переменные параметров и параметры оператора `catch`.
- Параметры типа.
- Пакеты.

В объявлениях классов и интерфейсов аннотации указываются перед ключевым словом `class` или `interface` следующим образом:

```
@Entity public class User { ... }
```

А в объявлениях переменных аннотации указываются перед типом переменной таким образом:

```
@SuppressWarnings("unchecked") List<User> users = ...;
public User getUser(@Param("id") String userId)
```

Параметр типа в обобщенном классе или методе может быть аннотирован следующим образом:

```
public class Cache<@Immutable V> { ... }
```

Пакет аннотируется в отдельном файле `package-info.java`. Этот файл содержит только операторы объявления и импорта пакета с предшествующими аннотациями, как показано ниже. Обратите внимание на то, что оператор `import` следует *после* оператора `package`, в котором объявляется пакет.

```
/**
  Документирующий комментарий на уровне пакета
*/
@GPL(version="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

НА ЗАМЕТКУ. Аннотации локальных переменных и пакетов отбрасываются при компилировании класса. Следовательно, они могут быть обработаны только на уровне исходного кода.

11.1.4. Аннотации в местах употребления типов

Аннотация в объявлении предоставляет некоторые сведения об объявляемом элементе кода. Так, в следующем примере кода аннотацией утверждается, что параметр `userId` объявляемого метода не является пустым:

```
public User getUser(@NonNull String userId)
```

НА ЗАМЕТКУ. Аннотация `@NonNull` является частью каркаса Checker Framework (**http://types.cs.washington.edu/checker-framework**). С помощью этого каркаса можно включать утверждения в прикладную программу, например, утверждение, что параметр не является пустым или относится к типу `String` и содержит регулярное выражение. В таком случае инструментальное средство статистического анализа проверит достоверность утверждений в данном теле исходного кода.

А теперь допустим, что имеется параметр типа `List<String>` и требуется каким-то образом указать, что все символьные строки не являются пустыми. Именно здесь и пригодятся аннотации в местах употребления типов. Такую аннотацию достаточно указать перед аргументом типа следующим образом:

```
List<@NonNull String>
```

Подобные аннотации можно указывать в следующих местах употребления типов:

- Вместе с аргументами обобщенного типа: `List<@NonNull String>`, `Comparator.<@NonNull String> reverseOrder()`.
- В любом месте массива: `@NonNull String[][] words` (элемент массива `words[i][j]` не является пустым), `String @NonNull [][] words` (массив `words` не является пустым), `String[] @NonNull [] words` (элемент массива `words[i]` не является пустым).
- В суперклассах и реализуемых интерфейсах: `class Warning extends @Localized Message`.
- В вызовах конструкторов: `new @Localized String(...)`.
- Во вложенных типах: `Map.@Localized Entry`.
- В приведении и проверках типов `instanceof`: `(@Localized String) text`, `if (text instanceof @Localized String)`. (Аннотации служат для употребления только внешними инструментальными средствами. Они не оказывают никакого влияния на поведение приведения и проверки типов `instanceof`.)
- В местах указания исключений: `public String read() throws @Localized IOException`.
- Вместе с метасимволами подстановки и ограничителями типов: `List<@Localized ? extends Message>`, `List<? Extends @Localized Message>`.
- В ссылках на методы и конструкторы: `@Localized Message::getText`.

Имеются все же некоторые методы употребления типов, где аннотации не допускаются. Ниже приведены характерные тому примеры.

```
@NonNull String.class // ОШИБКА: литерал класса не
                      // подлежит аннотированию!
import java.lang.@NonNull String; // ОШИБКА: импорт не
                                  // подлежит аннотированию!
```

Аннотации можно размещать до или после других модификаторов доступа вроде `private` и `static`. Обычно (хотя и не обязательно) аннотации в местах употребления

типов размещаются после других модификаторов доступа, тогда как аннотации в объявлениях — перед другими модификаторами доступа. Ниже приведены характерные тому примеры.

```
private @NonNull String text; // Аннотация в месте употребления типа
@Id private String userId; // Аннотация в объявлении переменной
```

НА ЗАМЕТКУ. Как поясняется в разделе 11.2, автор аннотации должен указать место, в котором может появиться конкретная аннотация. Если аннотация допускается как в объявлении переменной, так и в месте употребления типа, а применяется в объявлении переменной, то она указывается и в том и в другом месте. Рассмотрим в качестве примера следующее объявление метода:

```
public User getUser(@NonNull String userId)
```

Если аннотацию `@NonNull` можно применять как в параметрах, так и в местах употребления типов, то параметр **`userId`** аннотируется, а тип параметра обозначается как **`@NonNull String`**.

11.1.5. Явное указание получателей аннотаций

Допустим, что требуется аннотировать параметры, которые не изменяются методом, как показано ниже.

```
public class Point {
  public boolean equals(@ReadOnly Object other) { ... }
}
```

В таком случае инструментальное средство, обрабатывающее данную аннотацию, после анализа следующего вызова:

```
p.equals(q)
```

посчитает, что параметр `q` не изменился. А как насчет ссылки `p`? При вызове данного метода переменная получателя `this` привязывается к ссылке `p`. Но ведь переменная получателя `this` вообще не объявляется, а следовательно, она и не может быть аннотирована.

На самом деле эту переменную можно объявить с помощью редко употребляемой разновидности синтаксиса, чтобы ввести аннотацию следующим образом:

```
public class Point {
  public boolean equals(@ReadOnly Point this, @ReadOnly Object other) { ... }
}
```

Первый параметр в приведенном выше примере кода называется *параметром получателя*. Он должен непременно называться **`this`**. Его тип относится к тому классу, объект которого строится.

НА ЗАМЕТКУ. Параметром получателя можно снабдить *только* методы, но *не* конструкторы. По существу, ссылка **`this`** в конструкторе не является объектом данного типа до тех пор, пока конструктор не завершится. Напротив, аннотация, размещаемая в конструкторе, описывает строящийся объект.

Конструктору внутреннего класса передается другой скрытый параметр, а именно: ссылка на объект объемлющего класса. Этот параметр также можно указать явным образом, как показано ниже.

```
static class Sequence {
  private int from;
  private int to;

  class Iterator implements java.util.Iterator<Integer> {
    private int current;

    public Iterator(@ReadOnly Sequence Sequence.this) {
      this.current = Sequence.this.from;
    }
    ...
  }
  ...
}
```

Этот параметр именуется таким же образом, как и при ссылке на него: *ОбъемлющийКласс*.this. А его тип относится к объемлющему классу.

11.2. Определение аннотаций

Каждая аннотация должна быть объявлена в *интерфейсе аннотаций* с помощью синтаксиса @interface. Методы этого интерфейса соответствуют элементам аннотации. Например, аннотация Test блочного теста в JUnit определяется в следующем интерфейсе:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
  long timeout();
  ...
}
```

В объявлении @interface создается конкретный интерфейс Java. Инструментальные средства, обрабатывающие аннотации, получают объекты классов, реализующих интерфейс аннотаций. Когда, например, исполнитель текстов в инструментальном средстве JUnit получает объект класса, реализующего интерфейс Test, он просто вызывает метод timeout(), чтобы извлечь элемент установки времени ожидания из конкретной аннотации Test.

Аннотации Target и Retention являются *мета-аннотациями*. Они служат аннотациями к аннотации Test, обозначая места, где аннотация может произойти и где она доступна.

Значением мета-аннотации @Target служит массив объектов типа ElementType, обозначающих элементы, к которым можно применить аннотацию. В фигурных скобках можно указать любое количество типов элементов, как показано в следующем примере кода:

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

Все допустимые адресаты аннотаций перечислены в табл. 11.1. Компилятор проверяет, применяется ли аннотация только там, где это разрешено. Так, если аннотация `@BugReport` применяется к переменной, то во время компиляции возникает ошибка.

НА ЗАМЕТКУ. Аннотация без ограничения `@Target` может употребляться в любых объявлениях, но в не в параметрах и местах употребления типов. (Эти места были единственными допустимыми адресами аннотаций в первой версии Java, где поддерживались аннотации.)

В мета-аннотации `@Retention` указывается место, где аннотация может быть доступна. Этих мест может быть только три, как поясняется ниже.

1. `RetentionPolicy.SOURCE`. Аннотация доступна для процессов исходного кода, но не включается в файлы классов.
2. `RetentionPolicy.CLASS`. Аннотация включается в файлы классов, но виртуальная машина не загружает их. Этот вариант выбирается по умолчанию.
3. `RetentionPolicy.RUNTIME`. Аннотация доступна во время выполнения и через прикладной программный интерфейс API для рефлексии.

Таблица 11.1. Типы элементов для аннотации `@Target`

Тип элемента	Где применяется аннотация
`ANNOTATION_TYPE`	Объявления типов аннотаций
`PACKAGE`	Пакеты
`TYPE`	Классы (включая и перечисления) и интерфейсы (в том числе и типов аннотаций)
`METHOD`	Методы
`CONSTRUCTOR`	Конструкторы
`FIELD`	Переменные экземпляра (включая и константы перечислимого типа)
`PARAMETER`	Параметры методов и конструкторов
`LOCAL_VARIABLE`	Локальные переменные
`TYPE_PARAMETER`	Параметры типов
`TYPE_USE`	Места употребления типов

Примеры выбора всех трех перечисленных выше мест для доступа к аннотациям приведены далее в этой главе. Имеются и другие мета-аннотации, они перечисляются полностью далее, в разделе 11.3.

Чтобы задать значение по умолчанию для элемента аннотации, достаточно указать оператор `default` после метода, определяющего этот элемент, как выделено ниже полужирным.

```
public @interface Test {
  long timeout() default 0L;
  ...
}
```

В следующем примере кода демонстрируется, каким образом задается пустой массив и значение по умолчанию для аннотации:

```
public @interface BugReport {
  String[] reportedBy() default {};
    // Пустой массив по умолчанию
  Reference ref() default @Reference(id=0);
    // Значение по умолчанию для аннотации
  ...
}
```

ВНИМАНИЕ. Значения по умолчанию не хранятся вместе с аннотацией. Вместо этого они вычисляются динамически. Если изменить значение по умолчанию и перекомпилировать аннотированный класс, во всех аннотированных его элементах будет использовано новое значение по умолчанию, даже если файлы классов компилировались до изменения этого значения.

Интерфейсы аннотаций расширению не подлежат. Это означает, что для реализации интерфейсов нельзя предоставить конкретные классы. Вместо этого инструментальные средства обработки исходного кода и виртуальная машина генерируют классы и объекты-заместители по мере надобности.

11.3. Стандартные аннотации

В пакетах `java.lang`, `java.lang.annotation` и `javax.annotation` из прикладного программного интерфейса Java API определяется целый ряд интерфейсов аннотаций. Четыре из них относятся к мета-аннотациям, описывающим поведение интерфейсов аннотаций, а другие — к обычным аннотациям, служащим для аннотирования отдельных элементов в исходном коде. Все эти разновидности аннотаций перечислены в табл. 11.2, а подробнее они рассматриваются в двух последующих разделах.

Таблица 11.2. Стандартные аннотации

Интерфейс аннотаций	Где применяется	Назначение
`Override`	Методы	Проверяет, переопределяет ли данный метод соответствующий метод из суперкласса
`Deprecated`	Все объявления	Помечает элемент кода как не рекомендованный к употреблению
`SuppressWarnings`	Все объявления, кроме пакетов	Подавляет предупреждения данного типа
`SafeVarargs`	Методы и конструкторы	Утверждает, что пользоваться аргументами переменной длины безопасно
`FunctionalInterface`	Интерфейсы	Помечает интерфейс как функциональный с единственным абстрактным методом
`PostConstruct`	Методы	Метод должен быть вызван сразу же после построения
`PreDestroy`		или до удаления внедряемого объекта

Окончание табл. 11.2

Интерфейс аннотаций	Где применяется	Назначение
`Resource`	Классы и интерфейсы, методы, поля	Класс и интерфейс помечаются как ресурс, используемый повсеместно, а метод или поле — для внедрения зависимостей
`Resources`	Классы и интерфейсы	Обозначает массив ресурсов
`Generated`	Все объявления	Помечает элемент исходного кода как сформированный инструментальным средством
`Target`	Аннотации	Обозначает места, где может быть применена данная аннотация
`Retention`	Аннотации	Обозначает места, где может быть применена данная аннотация
`Documented`	Аннотации	Обозначает, что данная аннотация должна быть включена в документацию на аннотированные элементы кода
`Inherited`	Аннотации	Обозначает, что данная аннотация наследуется подклассом
`Repeatable`	Аннотации	Обозначает, что данную аннотацию можно применить несколько раз к одному и тому же элементу кода

11.3.1. Аннотации для компиляции

Аннотация `@Deprecated` может быть присоединена к любым элементам кода, применение которых больше не поощряется. Компилятор выдаст предупреждение, если в исходном коде будет обнаружен не рекомендованный к употреблению элемент. Эта аннотация имеет то же назначение, что и дескриптор `@deprecated` документирующей документации.

Аннотация `@Override` вынуждает компилятор проверять, что аннотируемый метод действительно переопределяет метод из суперкласса. Так, если объявляется следующий класс:

```
public class Point {
  @Override public boolean equals(Point other) { ... }
  ...
}
```

то компилятор известит об ошибке в связи с тем, что метод `equals()` не переопределяет одноименный метод `equals()` из класса `Object`, поскольку параметр этого метода относится к типу `Object`, а не к типу `Point`.

Аннотация `@SuppressWarnings` дает компилятору команду подавить предупреждения конкретного типа, как показано в следующем примере кода:

```
@SuppressWarnings("unchecked") T[] result =
   (T[]) Array.newInstance(cl, n);
```

Аннотация `@SafeVarargs` утверждает, что метод не нарушает свой параметр переменной длины (см. главу 6).

Аннотация `@Generated` предназначена для применения в инструментальных средствах генерирования кода. Любой генерируемый исходный код может быть аннотирован, чтобы отличать его от кода, написанного вручную. Например, в редакторе исходного текста можно скрыть генерируемый код, а в генераторе кода — удалить прежние версии генерируемого кода. Каждая аннотация должна содержать однозначный идентификатор генератора кода. Дополнительную строку с датой (в формате по стандарту ISO 8601) и строку комментариев указывать можно, но не обязательно:

```
@Generated(value="com.horstmann.generator",
    date="2015-01-04T12:08:56.235-0700");
```

В главе 3 был приведен пример употребления аннотации `@FunctionalInterface`. Она аннотирует преобразование адресатов лямбда-выражений, как показано ниже. Если в дальнейшем ввести в данный интерфейс еще один абстрактный метод, компилятор выдаст ошибку.

```
@FunctionalInterface
public interface IntFunction<R> {
  R apply(int value);
}
```

Разумеется, такие аннотации должны быть введены в интерфейсы, описывающие отдельные функции. Имеются и другие интерфейсы с единственным абстрактным методом (например, интерфейс `AutoCloseable`), по существу, не являющиеся функциями.

11.3.2. Аннотации для управления ресурсами

Аннотации `@PostConstruct` и `@PreDestroy` применяются в средах, управляющих сроком действия объектов, например, в веб-контейнерах и серверах приложений. Методы, помеченные этими аннотациями, должны вызываться сразу же после построения объекта или непосредственно перед его удалением.

Аннотация `@Resource` предназначена для внедрения ресурсов. В качестве примера рассмотрим веб-приложение, осуществляющее доступ к базе данных. Безусловно, доступ к информации в базе данных не должен быть жестко закодирован в данном веб-приложении. Вместо этого у веб-контейнера имеется свой пользовательский интерфейс для установки параметров подключения к базе данных, а также имя источника данных, определяемое в прикладном программном интерфейсе JNDI. Обращаться к этому источнику данных из веб-приложения можно следующим образом:

```
@Resource(name="jdbc/employeedb")
private DataSource source;
```

Когда строится объект, содержащий приведенную выше переменную экземпляра, веб-контейнер внедряет ссылку на источник данных. Это означает, что он устанавливает в переменной экземпляра объект типа `DataSource`, настраиваемый на имя `"jdbc/employeedb"`.

11.3.3. Мета-аннотации

Мета-аннотации `@Target` и `@Retention` упоминались ранее в разделе 11.2. А мета-документация `@Documented` предоставляет указание для инструментальных средств документирования вроде утилиты **javadoc**. Документируемые аннотации следует рассматривать как разновидность модификаторов (например, `private` или `static`), употребляемых для целей документирования. А остальные аннотации не следует включать в документацию.

Например, аннотация `@SuppressWarnings` не документируется. Если метод или поле содержит такую аннотацию, то особенности ее реализации малоинтересны читающему документацию на прикладной код. С другой стороны, аннотация `@FunctionalInterface` документируется, поскольку программисту важно знать, что аннотируемый ею интерфейс предназначен для описания функции. Пример документируемой аннотации приведен на рис. 11.1.

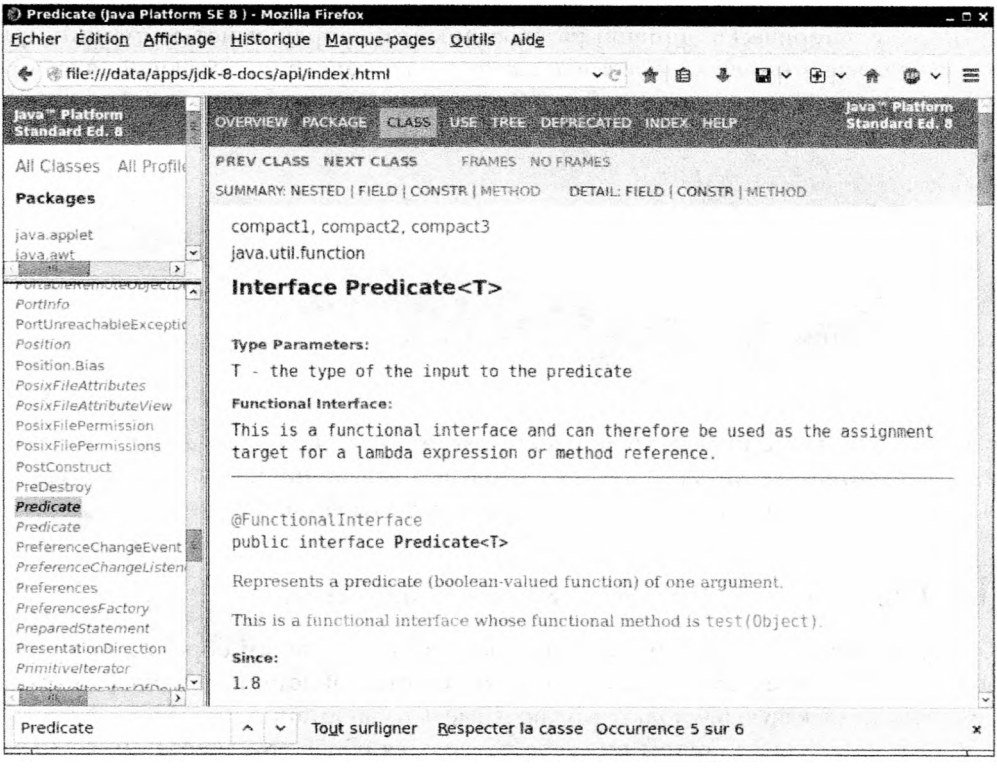


Рис. 11.1. Документируемая аннотация

Мета-аннотация `@Inherited` применяется только к аннотациям классов. Если в классе имеется наследуемая аннотация, то все его подклассы автоматически получают ту же самую аннотацию. Благодаря этому упрощается создание аннотаций, действующих аналогично маркерным интерфейсам (например, интерфейсу `Serializable`).

Допустим, что аннотация `@Persistent` объявляется с целью определить, что объекты класса могут быть сохранены в базе данных. В таком случае подклассы постоянных классов автоматически аннотируются как постоянные.

```
@Inherited @interface Persistent { }

@Persistent class Employee { . . . }
class Manager extends Employee { . . . }
    // Этот класс также имеет аннотацию @Persistent
```

Мета-аннотация `@Repeatable` позволяет применить одну и ту же аннотацию неоднократно. Допустим, что аннотация `@TestCase` повторяется. В таком случае ею можно воспользоваться следующим образом:

```
@TestCase(params="4", expected="24")
@TestCase(params="0", expected="1")
public static long factorial(int n) { ... }
```

По ряду исторических причин разработчикам повторяющейся аннотации пришлось предоставить *контейнерную аннотацию*, содержащую повторяющиеся аннотации в массиве. Ниже показано, каким образом определяется аннотация `@TestCase` и ее контейнер.

```
@Repeatable(TestCases.class)
@interface TestCase {
  String params();
  String expected();
}

@interface TestCases {
  TestCase[] value();
}
```

Всякий раз, когда пользователь предоставляет две или больше аннотации `@TestCase`, они автоматически заключаются в оболочку аннотации `@TestCases`. Это усложняет обработку аннотации, как будет показано в следующем разделе.

11.4. Обработка аннотаций во время выполнения

В приведенных до сих пор примерах было показано, каким образом аннотации вводятся в исходные файлы и как определяются типы аннотаций. А теперь настало время выяснить, какую же пользу можно извлечь из аннотаций.

В этом разделе на простом примере поясняется обработка аннотаций во время выполнения с использованием прикладного программного интерфейса API для рефлексии, рассматривавшегося в главе 4. Допустим, что требуется сократить затраты труда на реализацию методов типа **`toString`**. Можно, конечно, написать обобщенный метод `toString()`, используя рефлексию, чтобы учесть имена и значения всех переменных экземпляра. Но допустим, что этот процесс требуется специально настроить, чтобы не включать в него все переменные экземпляра или пропустить имена классов и переменных. Например, для класса `Point` более предпочтительной может

оказаться обозначение координат точки [5,10] вместо обозначения Point[x=5,y=10]. Разумеется, в данный процесс можно внести и другие усовершенствования, но мы не будем этого делать ради простоты примера. Самое главное — продемонстрировать в нем возможности процессора аннотаций.

Все классы, в которых требуется извлечь выгоду из данного процесса, следует снабдить аннотацией @ToString. Аннотировать следует и все переменные экземпляра, которые должны быть включены в данный процесс. Аннотация @ToString определяется следующим образом:

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ToString {
  boolean includeName() default true;
}
```

Ниже приведены аннотированные классы Point и Rectangle. При этом преследуется цель представить прямоугольник в виде символьной строки с параметрами Rectangle[[5, 10], width=20,height=30].

```
@ToString(includeName=false)
public class Point {
  @ToString(includeName=false) private int x;
  @ToString(includeName=false) private int y;
  ...
}

@ToString
public class Rectangle {
  @ToString(includeName=false) private Point topLeft;
  @ToString private int width;
  @ToString private int height;
  ...
}
```

Во время выполнения изменить реализацию метода toString() в отдельном классе нельзя. Вместо этого можно предоставить метод, способный отформатировать любой объект, обнаруживая и применяя аннотации @ToString, если они присутствуют.

Для обработки аннотаций служат следующие методы из интерфейса AnnotatedElement, который реализуется в классах рефлексии Class, Field, Parameter, Method, Constructor и Package:

```
T getAnnotation(Class<T>)
T getDeclaredAnnotation(Class<T>)
T[] getAnnotationsByType(Class<T>)
T[] getDeclaredAnnotationsByType(Class<T>)
Annotation[] getAnnotations()
Annotation[] getDeclaredAnnotations()
```

Как и остальные методы рефлексии, методы со словом Declared в их имени получают аннотации в самом классе, тогда как остальные методы включают наследуемые аннотации. В контексте аннотаций это означает аннотацию @Inherited, применяемую в суперклассе.

Если аннотация не является повторяющейся, для ее обнаружения следует вызвать метод `getAnnotation()`, как показано ниже.

```
Class cl = obj.getClass();
ToString ts = cl.getAnnotation(ToString.class);
if (ts != null && ts.includeName()) ...
```

Обратите внимание на то, что методу `getAnnotation()` передается объект класса для аннотации (в данном случае — объект `ToString.class`), а возвращается объект некоторого класса-заместителя, реализующего интерфейс `ToString`. Для получения значений элементов аннотации можно вызвать методы из этого интерфейса. Если же аннотация отсутствует, то метод `getAnnotation()` возвращает пустое значение `null`.

Дело несколько усложняется, если аннотация оказывается повторяющейся. Если вызвать метод `getAnnotation()` для поиска повторяющейся аннотации, которая на самом деле не повторялась, то и в этом случае может быть получено пустое значение `null`. Объясняется это тем, что повторяющиеся аннотации были заключены в оболочку контейнерной аннотации.

В данном случае следует вызвать метод `getAnnotationsByType()`, где просматривается контейнер и предоставляется массив повторяющихся аннотаций. Если бы присутствовала только одна аннотация, то она была бы получена в массиве единичной длины. Имея в своем распоряжении данный метод, можно вообще не беспокоиться о контейнерной аннотации.

Метод `getAnnotations()` получает все аннотации (любого типа), которыми аннотируется элемент кода, причем повторяющиеся аннотации заключаются в оболочку контейнеров. Ниже приведен пример реализации метода `toString()` с учетом аннотаций.

```
public class ToStrings {
  public static String toString(Object obj) {
    if (obj == null) return "null";
    Class<?> cl = obj.getClass();
    ToString ts = cl.getAnnotation(ToString.class);
    if (ts == null) return obj.toString();
    StringBuilder result = new StringBuilder();
    if (ts.includeName()) result.append(cl.getName());
    result.append("[");
    boolean first = true;
    for (Field f : cl.getDeclaredFields()) {
      ts = f.getAnnotation(ToString.class);
      if (ts != null) {
        if (first) first = false; else result.append(",");
        f.setAccessible(true);
        if (ts.includeName()) {
          result.append(f.getName());
          result.append("=");
        }
        try {
          result.append(ToStrings.toString(f.get(obj)));
        } catch (ReflectiveOperationException ex) {
          ex.printStackTrace();
        }
      }
```

```
    }
    result.append("]");
    return result.toString();
  }
}
```

Если класс аннотируется средствами интерфейса `ToString`, то в методе `toString()` перебираются поля этого класса и выводятся те из них, которые также аннотированы. Если же элемент `includeName` имеет логическое значение `true`, то имя класса или поля включается в результирующую символьную строку.

Следует иметь в виду, что данный метод вызывается рекурсивно. Всякий раз, когда объект принадлежит классу, который не аннотирован, вызывается его обычный метод `toString()` и рекурсия останавливается.

Это простой, но типичный пример применения прикладного программного интерфейса API для обработки аннотаций во время выполнения. Классы, поля и прочие элементы кода обнаруживаются с помощью рефлексии, а с целью извлечь аннотации вызывается метод `getAnnotation()` или `getAnnotationsByType()` для потенциально аннотированных элементов. И далее для получения значений отдельных элементов аннотации вызываются методы из соответствующего интерфейса аннотаций.

11.5. Обработка аннотаций на уровне исходного кода

В предыдущем разделе было показано, каким образом аннотации анализируются в выполняющейся программе. Еще одним примером применения аннотаций служит автоматическая обработка исходных файлов для получения дополнительного исходного кода, файлов конфигурации, сценариев и вообще всего, что можно сгенерировать.

Чтобы продемонстрировать внутренний механизм обработки аннотаций на уровне исходного кода, вернемся к примеру формирования методов типа `toString`. Но на этот раз они будут сформированы в исходном файле Java. Затем эти методы будут скомпилированы вместе с остальной частью программы и выполнены с максимальным быстродействием вместо применения рефлексии.

11.5.1. Процессоры аннотаций

Обработка аннотаций встроена в компилятор Java. Во время компиляции *процессоры аннотаций* можно вызывать по следующей команде:

```
javac -processor ИмяКлассаПроцессора1, ИмяКлассаПроцессора2,...
                 Исходные_файлы
```

Компилятор обнаруживает аннотации в исходных файлах. Каждый процессор аннотаций выполняется по очереди с учетом тех аннотаций, к которым он проявил интерес. Если процессор аннотаций создает новый исходный файл, то данный процесс повторяется. Как только все исходные файлы будут обработаны, они компилируются.

НА ЗАМЕТКУ. Процессор аннотаций может только формировать новые исходные файлы. Он не может изменять уже имеющиеся исходные файлы.

Процессор аннотаций реализует интерфейс Processor, как правило, расширяя класс AbstractProcessor. При этом нужно указать, какие именно аннотации поддерживаются процессором. В данном случае это следующие аннотации:

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class ToStringAnnotationProcessor extends AbstractProcessor {
  @Override
  public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment currentRound) {
  ...
  }
}
```

Процессору могут потребоваться конкретные типы аннотаций, метасимволы подстановки вроде "com.horstmann.*" (т.е. все аннотации из пакета com.horstmann и любых его подпакетов) или даже "*" (т.е. все аннотации вообще). Метод process() вызывается один раз на каждом цикле обработки со всеми аннотациями, обнаруженными в любых файлах в данном цикле, а также со ссылкой на интерфейс RoundEnvironment, содержащей сведения о текущем цикле обработки.

11.5.2. Прикладной программный интерфейс API модели языка

Для анализа аннотаций на уровне исходного кода служит прикладной программный интерфейс API *модели языка*. В отличие от прикладного программного интерфейса API для рефлексии, представляющий классы и методы на уровне виртуальной машины, прикладной программный интерфейс API модели языка позволяет анализировать программу на Java по правилам языка Java.

Компилятор получает дерево, узлами которого являются экземпляры классов, реализующих интерфейс javax.lang.model.element.Element и производные от него интерфейсы TypeElement, VariableElement, ExecutableElement и т.д. Они служат статическими аналогами классов рефлексии Class, Field/Parameter, Method/Constructor.

Не вдаваясь в подробности прикладного программного интерфейса API модели языка, ниже перечислены главные его особенности, о которых нужно знать для обработки аннотаций.

- Интерфейс RoundEnvironment предоставляет все элементы кода, помеченные конкретной аннотацией. Для этой цели вызывается следующий метод:

```
Set<? extends Element> getElementsAnnotatedWith(
                       Class<? extends Annotation> a)
```

- Эквивалентом интерфейса AnnotateElement для обработки аннотаций на уровне исходного кода является интерфейс AnnotatedConstruct. Для получения обычных или повторяющихся аннотаций из отдельного аннотированного класса служат следующие методы:

```
A getAnnotation(Class<A> annotationType)
A[] getAnnotationsByType(Class<A> annotationType)
```

- Интерфейс TypeElement представляет класс или интерфейс. А метод getEnclosedElements() получает список его полей и методов.

- В результате вызова метода `getSimpleName()` по ссылке типа `Element` или метода `getQualifiedName()` по ссылке типа `TypeElement` получается объект типа `Name`, который может быть преобразован в символьную строку методом `toString()`.

11.5.3. Генерирование исходного кода с помощью аннотаций

Вернемся к рассмотренному ранее примеру генерирования методов типа **`toString`**. Эти методы нельзя ввести в исходные классы. Ведь процессоры аннотаций способны производить только новые классы, а не изменять уже имеющиеся. Следовательно, все методы должны быть введены в служебный класс `ToStrings` следующим образом:

```
public class ToStrings {
  public static String toString(Point obj) {
    Сгенерированный код
  }
  public static String toString(Rectangle obj) {
    Сгенерированный код
  }
  ...
  public static String toString(Object obj) {
    return Objects.toString(obj);
  }
}
```

В данном случае применять рефлексию не требуется, и поэтому аннотируются методы доступа, но не поля, как показано ниже.

```
@ToString
public class Rectangle {
  ...
  @ToString(includeName=false) public Point getTopLeft()
      { return topLeft; }
  @ToString public int getWidth() { return width; }
  @ToString public int getHeight() { return height; }
}
```

И тогда процессор аннотаций должен сгенерировать следующий исходный код:

```
public static String toString(Rectangle obj) {
  StringBuilder result = new StringBuilder();
  result.append("Rectangle");
  result.append("[");
  result.append(toString(obj.getTopLeft()));
  result.append(",");
  result.append("width=");
  result.append(toString(obj.getWidth()));
  result.append(",");
  result.append("height=");
  result.append(toString(obj.getHeight()));
  result.append("]");
  return result.toString();
}
```

Шаблонный код выделен выше обычным шрифтом. Ниже приведен набросок метода, получающего метод `toString()` для класса с заданным параметром типа `TypeElement`.

```
private void writeToStringMethod(PrintWriter out, TypeElement te) {
  String className = te.getQualifiedName().toString();
  Вывести заголовок метода и объявление построителя символьных строк
  ToString ann = te.getAnnotation(ToString.class);
  if (ann.includeName()) Вывести код для ввода имени класса
  for (Element c : te.getEnclosedElements()) {
    ann = c.getAnnotation(ToString.class);
    if (ann != null) {
      if (ann.includeName()) Вывести код для ввода имени поля
        Вывести код для присоединения метода toString(obj.ИмяМетода())
    }
  }
  Вывести код для возврата символьной строки
}
```

А ниже приведен набросок метода `process()` из процессора аннотаций. В этом методе создается исходный файл для вспомогательного класса, а также выводится заголовок класса и по одному методу для каждого аннотируемого класса.

```
public boolean process(Set<? extends TypeElement> annotations,
       RoundEnvironment currentRound) {
  if (annotations.size() == 0) return true;
  try {
    JavaFileObject sourceFile =
           processingEnv.getFiler().createSourceFile(
                        "com.horstmann.annotations.ToStrings");
    try (PrintWriter out = new PrintWriter(sourceFile.openWriter())) {
      Вывести код для пакета и класса
      for (Element e : currentRound.getElementsAnnotatedWith(
                    ToString.class)) {
         if (e instanceof TypeElement) {
            TypeElement te = (TypeElement) e;
            writeToStringMethod(out, te);
      }
    }
    Вывести код для метода toString(Object)
    } catch (IOException ex) {
      processingEnv.getMessager().printMessage(
          Kind.ERROR, ex.getMessage());
    }
  }
  return true;
}
```

За более подробными сведениями обращайтесь к примерам кода, сопровождающего данную книгу. Следует, однако, иметь в виду, что метод `process()` вызывается в последующих циклах обработки аннотаций с пустым списком аннотаций. И тогда происходит немедленный возврат из данного метода, чтобы не создавать исходный файл дважды.

СОВЕТ. Чтобы просмотреть циклы обработки аннотаций, выполните команду `javac` с параметром `-XprintRounds`. В итоге на экран будет выведен результат, аналогичный следующему:

```
Round 1:
input files: {ch11.sec05.Point, ch11.sec05.Rectangle,
  ch11.sec05.SourceLevelAnnotationDemo}
  annotations: [com.horstmann.annotations.ToString]
  last round: false
Round 2:
  input files: {com.horstmann.annotations.ToStrings}
  annotations: []
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true
```

В данном примере было продемонстрировано, каким образом инструментальные средства могут собирать аннотации из исходных файлов для получения других файлов. Формируемые в итоге файлы совсем не обязательно должны быть исходными. Процессоры аннотаций могут сформировать дескрипторы XML-разметки, файлы свойств, сценарии командного процессора, документацию в формате HTML и пр.

НА ЗАМЕТКУ. Выше было показано, каким образом обрабатываются аннотации в исходных файлах и в выполняющейся программе. Третья возможность состоит в том, чтобы обрабатывать аннотации в файлах классов, что обычно делается по ходу их загрузки в виртуальную машину. Для обнаружения и вычисления аннотаций и перезаписи байт-кодов потребуется инструментальное средство вроде ASM (`http://asm.ow2.org/`).

Упражнения

1. Поясните, каким образом можно изменить метод `Object.clone()`, чтобы воспользоваться аннотацией `@Cloneable` вместо маркерного интерфейса `Cloneable`.
2. Если бы аннотации присутствовали в первых версиях Java, то интерфейс `Serializable`, безусловно, был бы снабжен аннотацией. Реализуйте аннотацию `@Serializable`. Выберите текстовый или двоичный формат для сохраняемости. Предоставьте классы для потоков ввода-вывода, чтения и записи, сохраняющих состояние объектов путем сохранения и восстановления всех полей, содержащих значения примитивных типов или же самих поддающихся сериализации. Не обращайте пока что внимание на циклические ссылки.
3. Повторите предыдущее упражнение, но позаботьтесь о циклических ссылках.
4. Введите аннотацию `@Transient` в механизм сериализации, действующий подобно модификатору доступа `transient`.

5. Определите аннотацию `@Todo`, содержащую сообщение, описывающее все, что требуется сделать. Определите процессор аннотаций, производящий из исходного файла список напоминаний о том, что требуется сделать. Предоставьте описание аннотируемого элемента кода и сообщение о том, что требуется сделать.

6. Преобразуйте аннотацию из предыдущего упражнения в повторяющуюся аннотацию.

7. Если бы аннотации существовали в первых версиях Java, они, скорее всего, выполняли бы роль утилиты **javadoc**. Определите аннотации `@Param`, `@Return` и т.д. и составьте из них элементарный HTML-документ с помощью процессора аннотаций.

8. Реализуйте аннотацию `@TestCase`, сформировав исходный файл, имя которого состоит из имени класса, где эта аннотация встречается, а также из имени `Test`. Так, если исходный файл `MyMath.java` содержит следующие строки:

```
@TestCase(params="4", expected="24")
@TestCase(params="0", expected="1")
public static long factorial(int n) { ... }
```

   то сформируйте исходный файл `MyMathTest.java` со следующими операторами:

```
assert(MyMath.factorial(4) == 24);
assert(MyMath.factorial(0) == 1);
```

   Можете допустить, что тестовые методы являются статическими и что элемент аннотации `params` содержит разделяемый запятыми список параметров соответствующего типа.

9. Реализуйте аннотацию `@TestCase` как динамическую и предоставьте инструментальное средство для ее проверки. И в этом случае можете допустить, что тестовые методы являются статическими. Можете также ограничиться умеренным набором параметров и возвращаемых типов, описываемых символьными строками в элементах аннотации.

10. Реализуйте процессор аннотаций `@Resource`, принимающий объект некоторого класса и обнаруживающий поля типа `String`, помечаемые аннотацией `@Resource(name="`*`URL`*`")`. Затем организуйте загрузку содержимого по заданному URL и внедрите строковую переменную с этим содержимым, используя рефлексию.

Глава 12

Прикладной программный интерфейс API даты и времени

В этой главе...

Время летит как стрела, и мы можем легко установить начальный момент, чтобы отсчитывать время вперед и назад. Так почему же так трудно обращаться со временем? Все дело в самих людях. Не проще ли было, если бы мы обращались друг к другу следующим образом: "Встретимся в 1371409200, только не опаздывай!" Но ведь нам нужно соотносить время с конкретным временем суток и года. Именно здесь и возникают трудности. В версии Java 1.0 имелся класс `Date`, реализация которого теперь кажется наивной, а большинство его методов стали нерекомендованными к употреблению, начиная с версии Java 1.1, где был внедрен класс `Calendar`. Безусловно, его прикладной программный интерфейс API не был совершенным, его экземпляры были изменяемыми, и в нем не учитывались потерянные секунды. Более совершенным оказался прикладной программный интерфейс API даты и времени, внедренный в версии Java 8. В нем были устранены недостатки прошлых реализаций, и можно надеяться, что он послужит нам еще немало времени. В этой главе будет показано, что именно делает расчеты времени столь неприятными и как подобные трудности разрешаются в прикладном программном интерфейсе API даты и времени.

Основные положения этой главы приведены ниже.

1. Объекты классов из пакета `java.time` неизменяемы.
2. Объект типа `Instant` представляет конкретный момент времени аналогично объекту типа `Date`.
3. В интерпретации Java каждый день состоит точно из 86400 секунд, т.е. без учета потерянных секунд.
4. Объект типа `Duration` определяет разность двух моментов времени.
5. Объект типа `LocalDateTime` не учитывает часовой пояс.
6. Методы из класса `TemporalAdjuster` выполняют типичные календарные вычисления, в том числе обнаружение первого вторника месяца.
7. Объект типа `ZonedDateTime` определяет момент времени в заданном часовом поясе аналогично объекту типа `GregorianCalendar`.
8. При смене часового пояса с учетом перехода на летнее время следует пользоваться классом `Period`, а не `Duration`.
9. Для форматирования и синтаксического анализа дат и моментов времени следует пользоваться классом `DateTimeFormatter`.

12.1. Временная шкала

По традиции основополагающей единицей отсчета времени является секунда, производная от вращения Земли вокруг своей оси. Полный оборот Земля совершает за 24 часа или 24 × 60 × 60 = 86400 секунд, и поэтому точное определение секунды кажется делом астрономических измерений. К сожалению, Земля испытывает незначительные колебания при вращении, что потребовало более точного определения секунды. И такое определение было сформулировано в 1967 году. Оно вполне согласуется с исторически сложившимся определением и в то же время основывается

на внутреннем свойстве атомов Цезия-133. С тех пор официальное время хранится в разветвленной сети атомных часов.

Нередко официальные часы-хранители времени синхронизируют абсолютное время с вращением Земли. Прежде официальные секунды подвергались незначительной коррекции, но с 1972 года стали периодически вводиться так называемые "потерянные" секунды. (Теоретически секунду можно было бы время от времени удалять, но этого так и не произошло.) В настоящее время снова ведутся дискуссии об изменении системы отсчета времени. Очевидно, что причиной тому служат потерянные секунды, и поэтому во многих вычислительных системах применяется так называемое "сглаживание" там, где время искусственно замедляется или ускоряется перед потерянной секундой, чтобы сохранить ровно 86400 секунд в сутках. Такой способ вполне работоспособен, поскольку местное время на компьютере отсчитывается не совсем точно, а компьютеры обычно синхронизируются с внешней службой времени.

В соответствии со спецификацией на прикладной программный интерфейс API для даты и времени в Java требуется временная шкала, которая

- имеет 86400 секунд в сутках;
- точно соответствует официальному времени в полдень каждого дня;
- близко соответствует ему в другое время суток точно определенным способом.

Благодаря этому язык Java может гибко подстраиваться к будущим изменениям в отсчете официального времени. В языке Java класс `Instant` представляет точку на временной шкале. Исходная точка отчета времени, называемая *эпохой*, произвольно задана в полдень 1 января 1970 года на нулевом меридиане, проходящем через Гринвичскую королевскую обсерваторию в Лондоне. Аналогичное соглашение принято и для отсчета времени в Unix/POSIX. Начиная с исходной точки отчета, время измеряется в секундах, как вперед, так и назад, с точностью до наносекунд, а каждые сутки составляют **86400** секунд. При отсчете времени назад значения типа `Instant` достигают миллиарда лет (т.е. минимальной точки `Instant.MIN` на временной шкале). И хотя этого недостаточно, чтобы выразить возраст вселенной (около 13,5 млрд лет), но должно быть достаточно для практических целей. Ведь миллиард лет назад Земля была покрыта льдом и населялась микроскопическими предшественниками современных растений и животных. А максимальная точка на временной шкале (`Instant.MAX`) соответствует 31 декабря 1000000000 года.

В результате вызова статического метода `Instant.now()` получается текущий момент времени. Два момента времени можно сравнить с помощью методов `equals()` и `compareTo()` обычным образом, чтобы использовать моменты времени как отметки времени.

Для определения разности двух моментов времени служит статический метод `Duration.between()`. В качестве примера ниже показано, как измерить текущее время выполнения алгоритма.

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

Объект типа Duration определяет промежуток между двумя моментами времени. Длительность промежутка типа Duration в обычных единицах измерения времени можно получить, вызвав метод toNanos(), toMillis(), toSeconds(), toMinutes(), toHours() или toDays().

Для внутреннего хранения промежутков времени требуется не только значение типа long. В частности, количество секунд хранится в значении типа long, а количество наносекунд в дополнительном значении типа int. Так, если требуется произвести расчеты времени с точностью до наносекунд во всем промежутке типа Duration, то для этой цели можно воспользоваться одним из методов, перечисленных в табл. 12.1. В противном случае достаточно вызвать метод toNanos(), чтобы произвести расчеты времени со значениями типа long.

НА ЗАМЕТКУ. Чтобы переполнилось значение типа **long**, потребуется количество наносекунд порядка 300 лет.

Таблица 12.1. Методы, выполняющие арифметические операции над моментами и промежутками времени

Метод	Описание
plus(), minus()	Добавляет или вычитает промежуток времени из данного момента времени типа **Instant** или промежутка времени типа **Duration**
plusNanos(), plusMillis(), plusSeconds(), plusMinutes(), plusHours(), plusDays()	Добавляют количество заданных единиц измерения времени к данному моменту времени типа **Instant** или промежутку времени типа **Duration**
minusNanos(), minusMillis(), minusSeconds(), minusMinutes(), minusHours(), minusDays()	Вычитают количество заданных единиц измерения времени из данного момента времени типа **Instant** или промежутка времени типа **Duration**
multipliedBy(), dividedBy(), negated()	Возвращают промежуток времени, получаемый умножением или делением данного промежутка времени типа **Duration** на заданное значение типа **long** или **-1**. Масштабировать можно только промежутки, но не моменты времени
isZero(), isNegative()	Проверяют, является ли данный промежуток времени типа **Duration** нулевым или отрицательным

Так, если требуется проверить, выполняется ли один алгоритм, по крайней мере, в десять раз быстрее, чем другой, достаточно выполнить следующие расчеты:

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster =
  timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
    // или timeElapsed.toNanos() * 10 < timeElapsed2.toNanos()
```

НА ЗАМЕТКУ. Классы **Instant** и **Duration** неизменяемы, и поэтому все методы вроде **multipliedBy()** или **minus()** возвращают новые экземпляры этих классов.

12.2. Местные даты

А теперь перейдем от абсолютного к обычному в обиходе времени. Такое время в прикладном программном интерфейсе Java API представлено двумя категориями *местного времени и даты* и *поясного времени*. Местное время и дата обозначают время суток или дату, но не связаны с часовым поясом. Примером местной даты служит 14 июня 1093 года (в этот день родился Алонсо Черч, изобретатель лямбда-вычислений). Эта дата не содержит ни время суток, ни часовой пояс, и поэтому она не соответствует точному моменту времени. С другой стороны, дата 16 июля 1969 года, 09:32:00 по восточному поясному времени (момент запуска космического корабля "Аполлон-11") представляет точный момент времени на временной шкале с учетом часового пояса.

Во многих расчетах времени учитывать часовые пояса не требуется, а иногда они могут даже мешать. Допустим, что требуется запланировать еженедельные совещания в 10:00. Если добавить 7 дней (т.е. **7 × 24 × 60 × 60** секунд) к последнему часовому поясу, то невольно можно пересечь границу, обозначающую переход на летнее или зимнее время, и тогда совещание состоится на час раньше или позже!

Именно по этой причине разработчики прикладного программного интерфейса API даты и времени рекомендуют не пользоваться поясным временем, кроме тех случаев, когда требуется представить экземпляры абсолютного времени. Дни рождения, праздники, сроки исполнения и прочие моменты времени лучше всего представить в виде местного времени и даты.

Объект типа `LocalDate` определяет местную дату с указанием года, месяца и дня месяца. Для построения этого объекта можно воспользоваться статическим методом `now()` или `of()`, как показано ниже.

```
LocalDate today = LocalDate.now(); // Текущая дата
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
    // Применяется перечисление Month
```

В отличие от нестандартных соглашений, принятых в Unix и классе `java.util.Date`, где отсчет месяцев начинается с нуля, а отсчет лет — с 1900 года, месяц года можно обозначить обычными числами. С другой стороны, для этой цели можно воспользоваться перечислением `Month`. В табл. 12.2 приведены наиболее употребительные методы для обращения с объектами типа `LocalDate`.

Таблица 12.2. Методы из класса `LocalDate`

Метод	Описание
`now()`, `of()`	Эти статические методы служат для построения объекта типа **`LocalDate`**, исходя из текущего времени или заданного года, месяца и дня
`plusDays()`, `plusWeeks()`, `plusMonths()`, `plusYears()`	Добавляют количество дней, недель, месяцев или лет к местной дате, представленной объектом типа **`LocalDate`**
`minusDays()`, `minusWeeks()`, `minusMonths()`, `minusYears()`	Вычитают количество дней, недель, месяцев или лет из местной даты, представленной объектом типа **`LocalDate`**

Окончание табл. 12.2

Метод	Описание
`plus()`, `minus()`	Добавляют или вычитают промежуток времени типа **Duration** или период времени типа **Period**
`withDayOfMonth()`, `withDayOfYear()`, `withMonth()`, `withYear()`	Возвращают новую местную дату в виде объекта типа **LocalDate** с днем месяца, днем года или года, измененного на заданное значение
`getDayOfMonth()`	Получает день месяца в пределах от **1** до **31**
`getDayOfYear()`	Получает день года в пределах от **1** до **366**
`getDayOfWeek()`	Получает день недели, возвращая значение из перечисления **DayOfWeek**
`getMonth()`, `getMonthValue()`	Получают в виде значения из перечисления **Month** или числа в пределах от **1** до **12**
`getYear()`	Получает год в пределах от **-999999999** до **999999999**
`Until()`	Получает период времени типа **Period** или количество заданных единиц времени типа **ChronoUnit** в промежутке между двумя датами
`isBefore()`, `isAfter()`	Сравнивает данную местную дату типа **LocalDate** с другой датой
`isLeapYear()`	Возвращает логическое значение **true**, если год оказывается високосным, т.е. если он делится на **4**, но не на **100** или **400**. Этот алгоритм применяется ко всем предыдущим годам, хотя он исторически неточный. (Високосные годы были введены в 46 году до н.э., а правила деления на **100** или **400** — при реформе григорианского календаря в 1582 году. На повсеместное распространение этой реформы понадобилось более 300 лет.)

Например, День программиста приходится на 256-й день года. Ниже показано, насколько просто рассчитывается этот день.

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255);
    // 13 сентября, но в високосный год — 12 сентября
```

Напомним, что разность двух моментов времени составляет промежуток типа `Duration`. Для местных дат ему соответствует период времени типа `Period`, который выражается количеством прошедших лет, месяцев или дней. Разумеется, для получения местной даты дня своего рождения в следующем году можно сделать вызов `birthday.plus(Period.ofYears(1))`. Но в високосный год в результате вызова `birthday.plus(Duration.ofDays(365))` вряд ли получится правильный результат.

Метод `until()` возвращает разность двух местных дат. Например, в результате следующего вызова:

```
independenceDay.until(christmas)
```

получается период времени в 5 месяцев и 21 день, что не очень удобно, поскольку количество дней в месяце варьируется. Поэтому для выявления количества дней лучше сделать следующий вызов:

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 дня
```

ВНИМАНИЕ. Вызов некоторых методов из табл. 12.2 мог бы привести к получению несуществующих дат. Так, если добавить один месяц к дате 31 января, то в конечном итоге не должна получиться дата 31 февраля. Вместо генерирования исключения эти методы возвращают последний достоверный день месяца. Например, в результате одного из следующих вызовов:

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

или

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

получается дата 29 февраля 2016.

Метод `getDayOfWeek()` возвращает день недели в виде соответствующего значения из перечисления `DayOfWeek`. В частности, понедельнику соответствует значение `DayOfWeek.MONDAY`, равное **1**, а воскресенью — значение `DayOfWeek.SUNDAY`, равное **7**. Например, в результате следующего вызова:

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

возвращается значение **1**. У перечисления `DayOfWeek` имеются служебные методы `plus()` и `minus()` для расчета дней недели по модулю **7**. Так, в результате вызова `DayOfWeek.SATURDAY.plus(3)` возвращается значение `DayOfWeek.TUESDAY`.

НА ЗАМЕТКУ. Выходные дни фактически приходятся на конец недели. В то же время в классе **`java.util.Calendar`** воскресенью соответствует значение **1**, а субботе — значение **7**.

Помимо класса `LocalDate`, имеются классы `MonthDay`, `YearMonth` и `Year` для описания частичных дат. Например, дата 25 декабря (с указанным годом) может быть представлена объектом класса `MonthDay`.

12.3. Корректоры дат

Для целей планирования нередко требуется рассчитать такие даты, как первый вторник каждого месяца. В классе `TemporalAdjusters` предоставляется целый ряд статических методов для общих видов коррекции дат. Результат выполнения метода коррекции дат передается методу `with()`. Например, первый вторник месяца может быть рассчитан следующим образом:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
    TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

Как всегда, метод `with()` возвращает новый объект типа `LocalDate`, не изменяя оригинал. Доступные корректоры дат перечислены табл. 12.3.

Имеется также возможность создать свой корректор дат, реализовав интерфейс `TemporalAdjuster`. В качестве примера ниже приведен корректор дат, предназначенный для расчета следующего дня недели.

Таблица 12.3. Доступные корректоры дат из класса `TemporalAdjusters`

Метод	Описание
`next(weekday)`, **`previous(weekday)`**	Возвращают следующую или предыдущую дату, приходящуюся на день недели, определяемый параметром ***weekday***
`nextOrSame(weekday)`, **`previousOrSame(weekday)`**	Возвращают следующую или предыдущую дату, приходящуюся на день недели, определяемый параметром ***weekday***, начиная с указанной даты
`dayOfWeekInMonth(n, weekday)`	Возвращает ***n***-й день недели в месяце, определяемый параметром ***weekday***
`lastInMonth(weekday)`	Возвращает последний день недели в месяце, определяемый параметром ***weekday***
`firstDayOfMonth()`, **`firstDayOfNextMonth()`**, **`firstDayOfNextYear()`**, **`lastDayOfMonth()`**, **`lastDayOfPreviousMonth()`**, **`lastDayOfYear()`**	Возвращают дату, обозначаемую в имени вызываемого метода

```
TemporalAdjuster NEXT_WORKDAY = w -> {
  LocalDate result = (LocalDate) w;
  do {
    result = result.plusDays(1);
  } while (result.getDayOfWeek().getValue() >= 6);
  return result;
};

LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Обратите внимание на то, что параметр лямбда-выражения относится к типу `Temporal`, и поэтому он должен быть приведен к типу `LocalDate`. Избежать этого приведения типов можно с помощью метода `ofDateAdjuster()`, ожидающего в качестве параметра лямбда-выражение типа `UnaryOperator<LocalDate>`, как показано ниже.

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjuster(w -> {
  LocalDate result = w; // Без приведения типов
  do {
    result = result.plusDays(1);
  } while (result.getDayOfWeek().getValue() >= 6);
  return result;
});
```

12.4. Местное время

Класс `LocalTime` представляет местное время суток, например 15:30:00. Экземпляр класса `LocalTime` можно получить с помощью метода `now()` или `of()` следующим образом:

```
LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30); // или LocalTime.of(22, 30, 0)
```

В табл. 12.4 перечислены методы, выполняющие общие операции с местным временем. В частности, методы `plus()` и `minus()` заключают в себе операции с местным временем в течение всех суток, как показано в следующей строке кода:

```
LocalTime wakeup = bedtime.plusHours(8); // Подъем в 6:30:00
```

Таблица 12.4. Методы из класса `LocalTime`

Метод	Описание
`now()`, `of()`	Эти статические методы строят объект типа **`LocalTime`**, представляющий местное время, исходя из текущего времени или из заданных часов, минут, а возможно, секунд и наносекунд
`plusHours()`, `plusMinutes()`, `plusSeconds()`, `plusNanos()`	Добавляют количество часов, минут, секунд или наносекунд к местному времени, представленному объектом типа **`LocalTime`**
`minusHours()`, `minusMinutes()`, `minusSeconds()`, `minusNanos()`	Вычитают количество часов, минут, секунд или наносекунд из местного времени, представленного объектом типа **`LocalTime`**
`plus()`, `minus()`	Добавляют или вычитают промежуток времени типа **`Duration`**
`withHour()`, `withMinute()`, `withSecond()`, `withNano()`	Возвращают новый объект типа **`LocalTime`**, представляющий местное время, где час, минута, секунда или наносекунда изменены на заданное значение
`getHour()`, `getMinute()`, `getSecond()`, `getNano()`	Получают час, минуту, секунду или наносекунду местного времени, представленного объектом типа **`LocalTime`**
`toSecondOfDay()`, `toNanoOfDay()`	Возвращают количество секунд или наносекунд в промежутке между полуночью и местным временем, представленным объектом типа **`LocalTime`**
`isBefore()`, `isAfter()`	Сравнивают данное местное время, представленное объектом типа **`LocalTime`**, с другим местным временем

НА ЗАМЕТКУ. В самом классе **`LocalTime`** местное время до и после полудня не разграничивается, поскольку эта обязанность возлагается на средство форматирования, как поясняется далее, в разделе 12.6.

Имеется также класс `LocalDateTime`, представляющий дату и время. Этот класс пригоден для хранения моментов времени в фиксированном часовом поясе, например, для планирования занятий или событий. Но если требуется произвести расчеты с учетом перехода на летнее время или поддерживать пользователей из разных часовых поясов, то для этих целей следует воспользоваться классом `ZonedDateTime`, рассматриваемым в следующем разделе.

12.5. Поясное время

Еще большую путаницу, чем неравномерность вращения Земли, в расчеты времени вносят часовые пояса, вероятно, потому, что они являются полностью изобретением

человечества. Люди во всем мире ведут отсчет от времени по Гринвичу, и поэтому одни обедают в 2:00, а другие в 22:00, исходя из потребностей своего желудка, а не времени суток. Так, в Китае обедают в разное время, поскольку на эту страну приходятся четыре условных часовых пояса. Эти пояса считаются условными, поскольку их границы неточны и смещаются, а переход на летнее и зимнее время еще больше усугубляет положение.

Как бы часовые пояса ни досаждали просвещенным, они являются непреложным фактом нашей жизни. Разрабатывая календарное приложение, нужно учитывать интересы людей, перемещающихся из одной страны в другую. Так, если требуется вовремя прибыть к 10:00 на конференцию в Нью-Йорке из Берлина, нужно правильно учесть местное время, определяя время отъезда.

В организации IANA (Internet Assigned Numbers Authority — Комитет по цифровым адресам в Интернете) ведется база данных всех известных в мире часовых поясов (`https://www.iana.org/time-zones`), обновляемая несколько раз в год. Большая часть обновлений относится к изменению правил перехода на летнее и зимнее время. База данных IANA применяется и в Java.

Каждому часовому поясу присваивается свой идентификатор, например `America/New_York` или `Europe/Berlin`. Чтобы выяснить все имеющиеся часовые пояса, достаточно вызвать метод `ZoneId.getAvailableIds()`. На момент написания данной книги насчитывалось более 600 идентификаторов часовых поясов.

Получая в качестве параметра идентификатор часового пояса, статический метод `ZoneId.of(id)` возвращает объект типа `ZoneId`. Этот объект можно использовать для преобразования объекта типа `LocalDateTime` в объект типа `ZonedDateTime`, вызвав метод `local.atZone(zoneId)`, или для построения объекта типа `ZonedDateTime`, вызвав статический метод `ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId)`, как показано в следующем примере кода:

```
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
    ZoneId.of("America/New_York"));
    // 1969-07-16T09:32-04:00[America/New_York]
```

Это конкретный момент времени. Чтобы получить объект типа `Instant`, достаточно сделать вызов `apollo11launch.toInstant()`. С другой стороны, если имеется конкретный момент времени, достаточно сделать вызов `instant.atZone(ZoneId.of("UTC"))`, чтобы получить объект типа `ZonedDateTime`, определяющий поясное время и дату по Гринвичу, или воспользоваться другим объектом типа `ZoneId`, чтобы получить дату и время в любом другом месте планеты.

НА ЗАМЕТКУ. Сокращение UTC обозначает Coordinated Universal Time (Всеобщее скоординированное время). Понятие UTC определяет время по Гринвичу без учета перехода на летнее или зимнее время.

Многие методы из класса `ZonedDateTime` похожи на методы из класса `LocalDateTime` (табл. 12.5). Большинство из них довольно просты, но переход на летнее и зимнее время несколько усложняет расчет поясного времени.

При переходе на летнее время стрелки часов переводятся на один час вперед. Что же произойдет, если рассчитать время, приходящееся на пропущенный час? Например, в 2013 году переход на летнее время в Центральной Европе произошел 31 марта в 2:00. Если попытаться рассчитать время в несуществующий момент 2:30 31 марта 2013 года, то на самом деле будет получено время 3:30:

```
ZonedDateTime skipped = ZonedDateTime.of(
  LocalDate.of(2013, 3, 31),
  LocalTime.of(2, 30),
  ZoneId.of("Europe/Berlin"));
  // Получается время 3:30 31 марта
```

С другой стороны, при переходе на зимнее время стрелки часов переводятся на один час назад, и в одно и то же местное время возникают два момента! При расчете времени в этом промежутке получается более ранний из этих двух моментов времени, как показано ниже.

```
ZonedDateTime ambiguous = ZonedDateTime.of(
    LocalDate.of(2013, 10, 27), // Переход на зимнее время
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
    // 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
    // 2013-10-27T02:30+01:00[Europe/Berlin]
```

Час спустя время будет показывать те же самые часы и минуты, но часовой пояс уже будет смещен. Следует также уделить внимание коррекции даты при пересечении границ перехода на летнее и зимнее время. Так, если требуется назначить совещание на следующей неделе, то не нужно вводить промежуток в семь дней, как в следующем примере кода:

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
    // Внимание! Этот код неверен при переходе на летнее время
```

Вместо этого лучше воспользоваться классом `Period` следующим образом:

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // Верно!
```

Таблица 12.5. Методы из класса `ZonedDateTime`

Метод	Описание
`now()`, `of()`, `ofInstant()`	Эти статические методы строят объект типа **`ZonedDateTime`**, представляющий поясное время и дату, исходя из текущего времени или года, месяца, дня, часа, минуты, секунды, наносекунды или местного времени (объекта типа **`LocalDate`**) и даты (объекта типа **`LocalTime`**), а также идентификатора часового пояса (объекта типа **`ZoneId`**) или текущего момента времени (объекта типа **`Instant`**) и идентификатора часового пояса (объекта типа **`ZoneId`**)

Окончание табл. 12.5

Метод	Описание
`plusDays()`, `plusWeeks()`, `plusMonths()`, `plusYears()`, `plusHours()`, `plusMinutes()`, `plusSeconds()`, `plusNanos()`	Добавляют количество соответствующих единиц измерения времени к поясному времени и дате, представленных объектом типа `ZonedDateTime`
`minusDays()`, `minusWeeks()`, `minusMonths()`, `minusYears()`, `minusHours()`, `minusMinutes()`, `minusSeconds()`, `minusNanos()`	Вычитают количество соответствующих единиц измерения времени из поясного времени и даты, представленных объектом типа `ZonedDateTime`
`plus()`, `minus()`	Добавляют или вычитают промежуток времени типа `Duration` или период времени типа `Period`
`withDayOfMonth()`, `withDayOfYear()`, `withMonth()`, `withYear()`, `withHour()`, `withMinute()`, `withSecond()`, `withNano()`	Возвращают новый объект типа `ZonedDateTime`, представляющий поясное время и дату, где час, минута, секунда или наносекунда изменены на заданное значение
`withZoneSameInstant()`, `withZoneSameLocal()`	Возвращают новый объект типа `ZonedDateTime`, представляющий в заданном часовом поясе тот же самый момент времени или то же самое локальное время
`getDayOfMonth()`	Получает день месяца в пределах от `1` до `31`
`getDayOfYear()`	Получает день года от `1` до `366`
`getDayOfWeek()`	Получает день недели, возвращая значение из перечисления `DayOfWeek`
`getMonth()`, `getMonthValue()`	Получают в виде значения из перечисления `Month` или числа в пределах от `1` до `12`
`getYear()`	Получает год в пределах от `-999999999` до `999999999`
`getHour()`, `getMinute()`, `getSecond()`, `getNano()`	Получают час, минуту, секунду или наносекунду поясного времени, представленного объектом типа `ZonedDateTime`
`getOffset()`	Получает смещение относительно времени UTC в виде экземпляра типа `ZoneOffset`. Смещение может изменяться в пределах от `12:00` до `+14:00`. У некоторых часовых поясов может быть дробное смещение. При переходе на летнее или зимнее время смещение изменяется
`toLocalDate()`, `toLocalTime()`, `toInstant()`	Получают локальное время, или дату, или же соответствующий момент времени
`isBefore()`, `isAfter()`	Сравнивает данное поясное время и дату типа `ZonedDateTime` с другим поясным временем и датой

ВНИМАНИЕ. Имеется также класс `OffsetDateTime`, представляющий время со смещением относительно времени UTC, но без учета правил смены часовых поясов. Этот класс предназначен для специального применения, требующего, в частности, отсутствия этих правил, в том числе и некоторых сетевых протоколов. Для получения поясного времени в удобочитаемом формате лучше пользоваться классом `ZonedDateTime`.

12.6. Форматирование и синтаксический анализ даты и времени

В классе `DateTimeFormatter` предоставляются следующие три вида средств форматирования для вывода значений даты и времени.

- Предопределенные стандартные средства форматирования, перечисленные в табл. 12.6.
- Средства форматирования с учетом региональных настроек.
- Средства форматирования по специальным шаблонам.

Чтобы воспользоваться одним из стандартных средств форматирования, достаточно вызвать его метод `format()` следующим образом:

```
String formatted = DateTimeFormatter.ISO_DATE_TIME.format(apollo11launch);
    // 1969-07-16T09:32:00-05:00[America/New_York]
```

Таблица 12.6. Предопределенные средства форматирования

Средство форматирования	Описание	Пример
`BASIC_ISO_DATE`	Год, месяц, день, смещение часового пояса без разделителей	`19690716-0500`
`ISO_LOCAL_DATE`, `ISO_LOCAL_TIME`, `ISO_LOCAL_DATE_TIME`	Разделители `-`, `:`, `T`	`1969-07-16`, `09:32:00`, `1969-07-16T09:32:00`
`ISO_OFFSET_DATE`, `ISO_OFFSET_TIME`, `ISO_OFFSET_DATE_TIME`	Аналогично `ISO_LOCAL_XXX`, но со смещением часового пояса	`1969-07-16-05:00`, `09:32:00-05:00`, `1969-07-16T09:32:00-05:00`
`ISO_ZONED_DATE_TIME`	Со смещением часового пояса и идентификатором часового пояса	`1969-07-16T09:32:00-05:00[America/New_York]`
`ISO_INSTANT`	Время в формате UTC, где `Z` обозначает идентификатор часового пояса	`1969-07-16T14:32:00Z`
`ISO_DATE`, `ISO_TIME`, `ISO_DATE_TIME`	Аналогично `ISO_OFFSET_DATE`, `ISO_OFFSET_TIME` и `ISO_ZONED_DATE_TIME`, но сведения о часовом поясе указываются дополнительно, хотя и не обязательно	`1969-07-16-05:00`, `09:32:00-05:00`, `1969-07-16T09:32:00-05:00[America/New_York]`
`ISO_ORDINAL_DATE`	Год и день года для местной даты типа `LocalDate`	`1969-197`
`ISO_WEEK_DATE`	Год, неделя и день недели для местной даты типа `LocalDate`	`1969-W29-3`
`RFC_1123_DATE_TIME`	Стандарт для отметок времени в электронной почте, кодируемых по стандарту RFC 822 и обновляемых четырьмя цифрами для обозначения года по стандарту RFC 1123	`Wed, 16 Jul 1969 09:32:00 -0500`

Стандартные средства предназначены в основном для форматирования машинно-читаемых отметок времени. А для представления дат и времени в удобочитаемом

виде служат средства форматирования с учетом региональных настроек. Они поддерживают четыре стиля форматирования даты и времени: SHORT, MEDIUM, LONG и FULL (табл. 12.7).

Таблица 12.7. Стили форматирования с учетом региональных настроек

Стиль	Дата	Время
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

Для создания такого средства форматирования служат статические методы ofLocalizedDate(), ofLocalizedTime() и ofLocalizedDateTime(). Ниже приведен характерный тому пример.

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
String formatted = formatter.format(apollo11launch);
    // July 16, 1969 9:32:00 AM EDT
```

В этих методах применяются региональные настройки, выбираемые по умолчанию. Чтобы выбрать другие региональные настройки, достаточно вызвать метод withLocale() следующим образом:

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo11launch);
    // 16 juillet 1969 09:32:00 EDT
```

У перечислений DayOfWeek и Month имеются методы getDisplayName() для получения наименований дней недели и месяца в разных форматах и региональных настройках, как показано ниже. Подробнее о региональных настройках речь пойдет в главе 13.

```
for (DayOfWeek w : DayOfWeek.values())
  System.out.print(
        w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + " ");
  // Выводит дни недели Mon Tue Wed Thu Fri Sat Sun на английском языке
```

НА ЗАМЕТКУ. Класс **java.time.format.DateTimeFormatter** служит для замены класса **java.util.DateFormat**. Если же требуется получить экземпляр последнего ради обратной совместимости, то следует вызвать метод **formatter.toFormat()**.

И наконец, можно создать свой формат даты, указав шаблон. Так, в следующей строке кода:

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

дата форматируется в форме **Wed 1969-07-16 09:32**. Каждая буква в шаблоне обозначает отдельное поле даты и времени, а количество повторений букв — конкретный

формат, выбираемый по правилам, которые не совсем ясны и, по-видимому, органично выработались со временем. Наиболее употребительные элементы шаблонов для форматирования даты и времени перечислены в табл. 12.8.

Таблица 12.8. Наиболее употребительные знаки в шаблонах для форматирования даты и времени

Константа перечислимого типа `ChronoField` или назначение	Перевод	Примеры
`ERA`	Эра	`G: AD, GGGG: Anno Domini, GGGGG: A`
`YEAR_OF_ERA`	Год эры	`yy: 69, yyyy: 1969`
`MONTH_OF_YEAR`	Месяц года	`M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J`
`DAY_OF_MONTH`	День месяца	`d: 6, dd: 06`
`DAY_OF_WEEK`	День недели	`e: 3, E: Wed, EEEE: Wednesday, EEEEE: W`
`HOUR_OF_DAY`	Час дня	`H: 9, HH: 09`
`CLOCK_HOUR_OF_AM_PM`	Полный час дня по часам до или после полудня	`K: 9, KK: 09`
`AMPM_OF_DAY`	Время суток до или после полудня	`a: AM`
`MINUTE_OF_HOUR`	Минута часа	`mm: 02`
`SECOND_OF_MINUTE`	Секунда минуты	`ss: 00`
`NANO_OF_SECOND`	Наносекунда секунды	`nnnnnn: 000000`
Идентификатор часового пояса		`VV: America/New_York`
Наименование часового пояса		`z: EDT, zzzz: Eastern Daylight Time`
Смещение часового пояса		`x: -04, xx: -0400, xxx: -04:00, XXX:` то же самое, но `Z` обозначает нуль
Локализованное смещение часового пояса		`O: GMT-4, OOOO: GMT-04:00`

Для синтаксического анализа значения даты и времени из символьной строки служит статический метод `parse()`, как показано ниже. В первом вызове этого метода используется стандартное средство форматирования `ISO_LOCAL_DATE`, а во втором вызове — специальное средство форматирования.

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11launch =
    ZonedDateTime.parse("1969-07-16 03:32:00-0400",
    DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

12.7. Взаимодействие с унаследованным кодом

Новый прикладной программный интерфейс Java API даты и времени должен обеспечивать нормальное взаимодействие с уже имеющимися классами. К их

числу относятся широко распространенные классы `java.util.Date`, `java.util.GregorianCalendar` и `java.sql.Date/Time/Timestamp`.

Класс `Instant` очень похож на класс `java.util.Date`. В версии Java 8 этот класс был дополнен двумя методами. В частности, метод `toInstant()` служит для преобразования типа `Date` в тип `Instant`, а статический метод `from()` — для обратного преобразования этих типов даты и времени.

Аналогично класс `ZonedDateTime` очень похож на класс `java.util.GregorianCalendar` и дополнен соответствующими методами преобразования в версии Java 8. В частности, метод `toZonedDateTime()` служит для преобразования типа `GregorianCalendar` в тип `ZonedDateTime`, а метод `from()` — для обратного преобразования этих типов даты и времени.

Еще один ряд преобразований предусмотрен для классов даты и времени из пакета `java.sql`. Кроме того, средство форматирования типа `DateTimeFormatter` можно передать унаследованному коду, где применяется класс `java.text.Format`. Все эти методы преобразования сведены в табл. 12.9.

Таблица 12.9. Методы взаимного преобразования классов из пакета `java.time` и унаследованного кода

Классы	Метод преобразования в унаследованный код	Метод преобразования из унаследованного кода
`Instant` ↔ `java.util.GregorianCalendar`	`Date.from(`*`instant`*`)`	`date.toInstant()`
`ZonedDateTime` ↔ `java.util.GregorianCalendar`	`GregorianCalendar.from(`*`zonedDateTime`*`)`	`cal.toZonedDateTime()`
`Instant` ↔ `java.sql.Timestamp`	`TimeStamp.from(`*`instant`*`)`	`timestamp.toInstant()`
`LocalDateTime` ↔ `java.sql.Timestamp`	`Timestamp.valueOf(`*`localDateTime`*`)`	`timeStamp.toLocalDateTime()`
`LocalDate` ↔ `java.sql.Date`	`Date.valueOf(`*`localDate`*`)`	`date.toLocalDate()`
`LocalTime` ↔ `java.sql.Time`	`Time.valueOf(`*`localTime`*`)`	`time.toLocalTime()`
`DateTimeFormatter` → `java.text.DateFormat`	`formatter.toFormat()`	Отсутствует
`java.util.TimeZone` ↔ `ZoneId`	`Timezone.getTimeZone(`*`id`*`)`	`timeZone.toZoneId()`
`java.nio.file.attribute.FileTime` ↔ `Instant`	`FileTime.from(`*`instant`*`)`	`fileTime.toInstant()`

Упражнения

1. Рассчитайте дату, на которую приходится День программиста, не пользуясь методом `plusDays()`.
2. Что произойдет, если добавить один год, четыре года, четыре раза по одному году при вызове метода `LocalDate.of(2000, 2, 29)`?
3. Реализуйте метод `next()`, принимающий в качестве параметра местную дату типа `Predicate<LocalDate>` и возвращающий корректор дат, который

получает следующую дату выполнения предиката. Например, в результате приведенного ниже вызова вычисляется следующий рабочий день.

```
today.with(next(w -> getDayOfWeek().getValue() < 6))
```

4. Напишите программу, выполняющую такие же функции, как и команда **cal** в Unix, выводящая календарь на текущий месяц. Например, по команде **java Cal 3 2013** должен быть выведен календарь на март 2013 года, где 1 марта приходится на пятницу, как показано ниже. (Обозначьте каким-нибудь образом в календаре конец каждой недели.)

```
                        1     2     3
 4     5     6     7     8     9    10
11    12    13    14    15    16    17
18    19    20    21    22    23    24
25    26    27    28    29    30    31
```

5. Напишите программу, выводящую количество дней, которые вы прожили.
6. Выведите перечень всех пятниц, которые пришлись на 13-е число в XX веке.
7. Реализуйте класс `TimeInterval`, представляющий промежуток времени, пригодный для календарных событий (например, совещаний, которые должны проходить в назначенный день с 10:00 до 11:00). Предоставьте метод, чтобы проверить, не перекрываются ли два промежутка времени.
8. Получите смещения текущей даты во всех поддерживаемых часовых поясах для текущего момента времени, преобразовав результат вызова метода `ZoneId.getAvailableZoneIds()` в поток данных и используя потоковые операции.
9. Используя снова потоковые операции, выявите все часовые пояса, смещения которых не кратны полному часу.
10. Самолет авиарейсом из Лос-Анджелеса во Франкфурт вылетает в 3:05 по полудни местного времени и находится в полете 10 часов 50 минут. Когда он прибывает во Франкфурт? Напишите программу, способную выполнять подобные расчеты времени.
11. Самолет обратным авиарейсом из Франкфурта вылетает в 14:05 и прибывает в Лос-Анджелес в 16:40. Сколько времени длится полет? Напишите программу, способную выполнять подобные расчеты времени.
12. Напишите программу, разрешающую затруднение, описанное в начале раздела 12.5. Проанализируйте ряд назначенных мероприятий в разных часовых поясах и предупредите пользователя о тех из них, которые должны состояться в течение следующего часа по местному времени.

Глава 13

Интернационализация

В этой главе...

Программным обеспечением, которое вы разрабатываете, могут заинтересоваться не только в вашей стране, но и за рубежом. Некоторые программисты считают, что для интернационализации их прикладных программ достаточно обеспечить поддержку Юникода и перевести сообщения в пользовательском интерфейсе на местный язык. Но, как будет показано в этой главе, интернационализация прикладных программ требует много большего. Время, даты, денежные единицы и даже числа по-разному форматируются в разных частях света. В этой главе поясняется, как пользоваться доступными в Java средствами интернационализации, чтобы представлять и принимать информацию в прикладных программах таким образом, чтобы она была понятна пользователям, где бы они ни проживали. А в конце этой главы дается краткий обзор прикладного программного интерфейса Java Preferences API для хранения пользовательских глобальных параметров.

Основные положения этой главы приведены ниже.

1. Если прикладная программа предназначается для международного использования, то для ее интернационализации недостаточно перевода одних только сообщений. В частности, форматирование чисел и дат заметно отличается в разных частях света.
2. Региональные настройки описывают язык и глобальные параметры форматирования для пользователей из отдельных стран и регионов мира.
3. В классах `NumberFormat` и `DateTimeFormat` форматирование чисел, денежных сумм, дат и моментов времени осуществляется с учетом региональных настроек.
4. Средствами класса `MessageFormat` можно форматировать символьные строки с метками-заполнителями, каждая из которых может иметь свой формат.
5. Для сортировки символьных строк с учетом региональных настроек служит класс `Collator`.
6. Класс `ResourceBundle` позволяет манипулировать локализованными символьными строками и объектами с учетом разных региональных настроек.
7. Класс `Preferences` служит для хранения пользовательских глобальных параметров независимо от конкретной платформы.

13.1. Региональные настройки

Очевидным отличием любой прикладной программы, адаптированной для международного использования, является язык пользовательского интерфейса. Но у нее имеется немало других менее заметных отличий. Например, числа в английском и немецком представлении форматируются совсем иначе. Так, следующее число в английском написании:

```
123,456.78
```

должно быть представлено в немецком написании следующим образом:

```
123.456,78
```

Как видите, назначение десятичной точки и разделяющей запятой в обоих случаях совершенно разное. Аналогичные расхождения имеются и в представлении дат. Так, в США даты отображаются в формате месяц/день/год, в Германии — в более удобном формате день/месяц/год, тогда как в Китае — в формате год/месяц/день. Следовательно, следующая дата в американском написании:

```
3/22/61
```

должна быть представлена немецкому пользователю следующим образом:

```
22.03.1961
```

Если наименования месяцев указываются явно, то отличия в их написании на разных языках становятся еще более заметными. Например, следующая дата на английском языке:

```
March 22, 1961
```

должна быть представлена на немецком языке следующим образом:

```
22. März 1961
```

Региональные настройки определяют язык и географическое место пользователя, что дает средствам форматирования возможность принимать во внимание пользовательские глобальные параметры. В последующих разделах будет показано, каким образом указываются региональные настройки и как они устанавливаются в программах на Java.

13.1.1. Указание региональных настроек

Региональные настройки содержат следующие составляющие.

1. Язык, обозначаемый двумя или тремя строчными буквами, например, **en** (английский), **de** (немецкий) или **zh** (китайский). Наиболее употребительные коды языков перечислены в табл. 13.1.
2. Дополнительно письмо, обозначаемое четырьмя буквами, первая из которых является заглавной, например **Latn** (латынь), **Cyrl** (кириллица) или **Hant** (традиционные китайские иероглифы). Это удобно, поскольку в ряде языков (например, в сербском) употребляется как латынь, так и кириллица, а некоторые китайские пользователи предпочитают традиционные иероглифы упрощенным.
3. Дополнительно страна или регион, обозначаемые двумя прописными буквами или тремя цифрами, например **US** (США) или **CH** (Швейцария). Наиболее употребительные коды стран перечислены в табл. 13.2.
4. Дополнительно *вариант*.
5. Дополнительно расширение. Расширения описывают локальные установки для календарей (например, японского календаря), чисел (тайских цифр вместо арабских) и т.д. Некоторые из этих расширений определены в стандарте на

Юникод. Расширения начинаются с обозначения **u-** и продолжаются двухбуквенным кодом, указывающим назначение расширения: **ca** — для календаря, **nu** — для чисел и т.д. Например, расширение **u-nu-thai** обозначает употребление тайских цифр. Другие расширения совершенно произвольны и начинаются с обозначения **x-**, например **x-java**.

НА ЗАМЕТКУ. Варианты редко употребляются в настоящее время. Раньше употреблялся "новонорвежский" вариант норвежского языка, но теперь он выражается отдельным кодом языка **nn**. А прежние варианты японского императорского календаря и тайских цифр теперь выражаются как расширения.

Правила для региональных настроек сформулированы в памятной записке BCP 47 "Best Current Practices" (Передовые современные методики) Рабочей группы инженерной поддержки Интернета (Internet Engineering Task Force; `http://tools.ietf.org/html/bcp47`). Более доступное краткое изложение этих правил можно найти по адресу `www.w3.org/International/articles/language-tags`.

НА ЗАМЕТКУ. Коды языков и стран кажутся на первый взгляд выбранными несколько произвольно, поскольку некоторые из них происходят от местных языков. Так, Deutsch по-немецки означает "немецкий", а zhongwen по-китайски (в латинской транскрипции) — "китайский", отсюда и коды этих языков **de** и **zh** соответственно. Швейцария обозначается кодом **CH**, происходящим об латинского термина *Confoederatio Helvetica*, означающего "Швейцарская конфедерация".

Таблица 13.1. Наиболее употребительные коды языков

Язык	Код	Язык	Код
Китайский	zh	Японский	ja
Датский	da	Корейский	ko
Голландский	du	Норвежский	no
Английский	en	Португальский	pt
Французский	fr	Испанский	es
Финский	fi	Шведский	sv
Итальянский	it	Турецкий	tr

Таблица 13.2. Наиболее употребительные коды стран

Страна	Код	Страна	Код
Австрия	AT	Япония	JP
Бельгия	BE	Корея	KR
Канада	CA	Нидерланды	NL
Китай	CN	Норвегия	NO
Дания	DK	Португалия	PT
Финляндия	FI	Испания	ES

Окончание табл. 13.2

Страна	Код	Страна	Код
Германия	**DE**	Швеция	**SE**
Великобритания	**GB**	Швейцария	**CH**
Греция	**GR**	Тайвань	**TW**
Ирландия	**IE**	Турция	**TR**
Италия	**IT**	США	**US**

Региональные настройки описываются дескрипторами в виде символьных строк, где элементы региональных настроек указываются через дефис, например, **en-US** для США или **de-DE** для Германии. А в Швейцарии приняты четыре официальных языка (немецкий, французский, итальянский и реторомanский), поэтому для немецкоязычного пользователя из Швейцарии потребуются региональные настройки **de-CH**, соблюдающие правила немецкого языка, но выражающие денежные суммы в швейцарских франках, а не в евро. Если же указать только язык (**de**), то такие региональные настройки нельзя использовать для выражения характерных для конкретной страны особенностей вроде представления денежных сумм в местной валюте.

Построить объект типа `Locale` из символьной строки дескриптора можно следующим образом:

```
Locale usEnglish = Locale.forLanguageTag("en-US");
```

Метод `toLanguageTag()` возвращает дескриптор языка из заданных региональных настроек. Так, в результате вызова `Locale.US.toLanguageTag()` возвращается символьная строка `"en-US"`.

Ради удобства для различных стран заранее определены следующие объекты региональных настроек:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Ряд заранее определенных региональных настроек обозначают только язык, но не страну или регион, как показано ниже. И наконец, статический метод `getAvailable Locales()` возвращает массив всех региональных настроек, известных виртуальной машине.

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

13.1.2. Региональные настройки по умолчанию

Статический метод `getDefault()` из класса `Locale` первоначально получает региональные настройки по умолчанию, хранящиеся в локальной операционной системе. В некоторых операционных системах пользователям разрешается указывать другие региональные настройки для вывода сообщений и других целей форматирования. Например, франкоязычному пользователю, проживающему в США, могут быть предоставлены меню на французском языке, но денежные суммы будут выражаться в долларах США.

Чтобы получить эти глобальные параметры, достаточно сделать следующие вызовы:

```
Locale displayLocale = Locale.getDefault(Locale.Category.DISPLAY);
Locale formatLocale = Locale.getDefault(Locale.Category.FORMAT);
```

НА ЗАМЕТКУ. В Unix отдельные региональные настройки можно указать для чисел, денежных единиц и дат, установив переменные окружения **C_NUMERIC**, **LC_MONETARY** и **LC_TIME** соответственно. Но в Java эти переменные не принимаются во внимание.

СОВЕТ. Для целей тестирования в прикладной программе может потребоваться переход к региональным настройкам по умолчанию. Для этого при запуске программы следует предоставить свойства языка и региона. В качестве примера региональные настройки по умолчанию для немецкоязычных пользователей из Швейцарии устанавливаются по следующей команде:

```
java -Duser.language=de -Duser.country=CH MainClass
```

Имеется также возможность изменить письмо и вариант. А для региональных настроек отображения и форматирования можно сделать отдельные установки, например **-Duser.script.display=Hant**.

Чтобы изменить региональные настройки виртуальной машины по умолчанию, достаточно сделать один из приведенных ниже вызовов. В результате первого вызова изменяются региональные настройки, возвращаемые методом `Locale.getDefault()`, а в результате второго вызова — для всех категорий.

```
Locale.setDefault(newLocale);
Locale.setDefault(category, newLocale);
```

13.1.3. Отображаемые имена

Допустим, что требуется разрешить пользователю выбирать среди ряда региональных настроек, но не отображать непонятные ему символьные строки дескрипторов. Метод `getDisplayName()` возвращает символьную строку, описывающую региональные настройки в форме, удобной для представления пользователю:

```
German (Switzerland)
```

Но дело в том, что отображаемое имя выводится на языке региональных настроек по умолчанию, что может оказаться неприемлемым. Так, если пользователь уже выбрал немецкий язык в качестве предпочтительного, то ему хотелось быть в дальнейшем получать информацию на немецком языке. Для этого региональные настройки на немецком языке следует передать соответствующему методу в виде параметра. Так, в результате выполнения следующего фрагмента кода:

```
Locale loc = Locale.forLanguageTag("de-CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

выводится такая символьная строка:

```
Deutsch (Schweiz)
```

Этот пример наглядно показывает, для чего нужны объекты типа `Locale`. Их можно передать методам, выводящим текстовую информацию пользователям с учетом разных региональных настроек. Многие тому примеры будут приведены в последующих разделах.

13.2. Форматы чисел

Класс `NumberFormat` из пакета `java.text` предоставляет для средств форматирования три следующих фабричных метода, способных форматировать числа и производить их синтаксический анализ: `getNumberInstance()`, `getCurrencyInstance()` и `getPercentInstance()`. В качестве примера ниже показано, как отформатировать денежную сумму в немецком представлении.

```
Locale loc = Locale.GERMANY;
NumberFormat formatter = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = formatter.format(amt);
```

Выполнение приведенного выше кода дает следующий результат:

```
123.456,78€
```

Обратите внимание на знак денежной единицы €, размещаемый в конце символьной строки выводимого результата, а также на обратный порядок употребления десятичной точки и разделяющей запятой по сравнению с принятым в американском представлении денежных сумм. С другой стороны, чтобы прочитать число, введенное

или сохраненное с учетом конкретных региональных настроек, можно воспользоваться методом parse() следующим образом:

```
String input = ...;
NumberFormat formatter = NumberFormat.getNumberInstance();
    // получить средство форматирования чисел для региональных
    // настроек форматирования по умолчанию
Number parsed = formatter.parse(input);
double x = parsed.doubleValue();
```

Метод parse() возвращает объект абстрактного типа Number. Это может быть объект-оболочка типа Double или Long, в зависимости от того, оказалось ли синтаксически проанализированное число в формате с плавающей точкой. Если это не имеет особого значения, то можно просто вызвать метод doubleValue() из класса Number, чтобы извлечь число из объекта-оболочки.

Если текст для представления числа оказывается в неправильной форме, то данный метод генерирует исключение типа ParseException. Например, начальные нули в символьной строке, содержащей анализируемое число, не допускаются. Для их удаления следует вызвать метод trim(). Но любые символы, следующие после числа в символьной строке, просто игнорируются, а исключение не генерируется.

13.3. Денежные единицы

Чтобы отформатировать денежную сумму в местной валюте, можно воспользоваться методом NumberFormat.getCurrencyInstance(). Но этот метод не очень удобен, поскольку он возвращает средство форматирования для единственной денежной единицы. Допустим, что для американского заказчика требуется подготовить счет-фактуру, где одни суммы должны быть указаны в долларах США, а другие — в евро. Для этой цели нельзя воспользоваться двумя разными средствами форматирования, как показано ниже.

```
NumberFormat dollarFormatter =
       NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter =
      NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

В итоге счет-фактура приобретет очень странный вид, поскольку одни денежные суммы будут отформатированы в нем как **$100,000**, а другие — как **100.000€**. (Обратите внимание на употребление десятичной точки вместо запятой в денежной сумме, выражаемой в евро.)

Вместо этого можно воспользоваться классом Currency, чтобы управлять денежными единицами, употребляемыми в средствах форматирования. Для этого достаточно получить объект типа Currency, передав в качестве параметра идентификатор денежной единицы статическому методу Currency.getInstance(). Наиболее употребительные идентификаторы денежных единиц приведены в табл. 13.3. Статический метод Currency.getAvailableCurrencies() получает множество типа Set<Currency> с денежными единицами, известными виртуальной машине.

Итак, получив объект типа Currency, можно вызвать метод setCurrency() для средства форматирования. В следующем примере кода демонстрируется, каким образом денежные суммы в евро форматируются для американского заказчика:

```
NumberFormat formatter = NumberFormat.getCurrencyInstance(Locale.US);
formatter.setCurrency(Currency.getInstance("EUR"));
System.out.println(formatter.format(euros));
```

Если требуется отобразить локализованные наименования или знаки денежных единиц, достаточно вызвать следующие методы:

```
getDisplayName()
getSymbol()
```

Эти методы возвращают символьные строки, исходя из региональных настроек отображения по умолчанию. Но в качестве параметра им можно передать другие региональные настройки.

Таблица 13.3. Наиболее употребительные идентификаторы денежных единиц

Денежная единица	Идентификатор	Денежная единица	Идентификатор
Доллар США	USD	Китайский женьминьби (юань)	CNY
Евро	EUR	Индийская рупия	INR
Британский фунт	GBP	Российский рубль	RUB
Японская йена	JPY	Швейцарский франк	CHF

13.4. Форматирование даты и времени

При форматировании даты и времени решаются следующие вопросы, связанные с региональными настройками.

1. Наименования месяцев и дней недели должны быть представлены на местном языке.
2. Для порядка указания года, месяца и дня должны быть отдельные локальные установки.
3. Для представления дат григорианский календарь не может быть локальной установкой.
4. Необходимо учитывать часовой пояс географического места.

Для форматирования даты и времени следует использовать класс DateTimeFormatter из пакета java.time.format, а не устаревший класс java.util.DateFormat. При этом нужно решить, нужна ли только дата, время или и то и другое. Затем выбирается один из четырех стилей форматирования, перечисленных в табл. 13.4. Если форматируются дата и время, эти стили можно выбрать отдельно.

Таблица 13.4. Стили форматирования даты и времени с учетом региональных настроек

Стиль	Дата	Время
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	

Далее получается средство форматирования следующим образом:

```
FormatStyle style = ...; // Один из стилей форматирования
                         // FormatStyle.SHORT, FormatStyle.MEDIUM, ...
DateTimeFormatter dateFormatter =
        DateTimeFormatter.ofLocalizedDate(style);
DateTimeFormatter timeFormatter =
        DateTimeFormatter.ofLocalizedTime(style);
DateTimeFormatter dateTimeFormatter =
        DateTimeFormatter.ofLocalizedDateTime(style);
        // или DateTimeFormatter.ofLocalizedDateTime(style1, style2)
```

В этих средствах форматирования используются текущие региональные настройки форматов даты и времени. Чтобы выбрать другие региональные настройки, достаточно вызвать метод `withLocale()` следующим образом:

```
DateTimeFormatter dateFormatter =
    DateTimeFormatter.ofLocalizedDate(style).withLocale(locale);
```

Теперь можно отформатировать местную дату (объект типа `LocalDate`), местное время и дату (объект типа `LocalDateTime`), местное время (объект типа `LocalTime`) или поясное время и дату (объект типа `ZonedDateTime`), как показано ниже.

```
ZonedDateTime appointment = ...;
String formatted = formatter.format(appointment);
```

Для синтаксического анализа символьной строки, содержащей дату и время, используется статический метод `parse()` из класса `LocalDate`, `LocalDateTime`, `LocalTime` или `ZonedDateTime`, как показано ниже. При неудачном исходе синтаксического анализа символьной строки генерируется исключение типа `DateTimeParseException`.

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

ВНИМАНИЕ. Методы **`parse()`** из упомянутых выше классов непригодны для синтаксического анализа данных, вводимых пользователем, — по крайней мере, для предварительной обработки. Например, средство форматирования даты и времени в кратком стиле для США способно проанализировать символьную строку **`"9:32 AM"`**, но не строку **`"9:32AM"`** или **`"9:32 am"`**.

ВНИМАНИЕ. Средства форматирования дат подвергают синтаксическому анализу несуществующие даты вроде 31 ноября, корректируя их по последней дате в данном месяце.

Иногда в календарном приложении требуется отображать, например, только наименования дней недели и месяцы. С этой целью можно вызвать метод `getDisplayName()` из перечислений `DayOfWeek` и `Month`, как показано ниже.

```
for (Month m : Month.values())
  System.out.println(m.getDisplayName(textStyle, locale) + " ");
```

Стили форматирования текста перечислены в табл. 13.5. Стили типа `STANDALONE` служат для отображения за пределами форматируемой даты. Например, январь по-фински обозначается как "tammikuuta" в самой дате, но как "tammikuu" за ее пределами или отдельно.

Таблица 13.5. Стили форматирования текста, представленные константами из перечисления `java.time.format.TextStyle`

Стиль	Пример
`FULL` / `FULL_STANDALONE`	January
`SHORT` / `SHORT_STANDALONE`	Jan
`NARROW` / `NARROW_STANDALONE`	J

13.5. Сортировка и нормализация

Как известно, для сравнения символьных строк служит метод `compareTo()` из класса `String`. К сожалению, этот метод не совсем годится для взаимодействия с пользователями. В методе `compareTo()` применяются строковые значения в кодировке UTF-16, что приводит к абсурдным результатам, даже на английском языке. Например, следующие пять символьных строк упорядочиваются по результатам сортировки методом `compareTo()` таким образом:

```
Athens
Zulu
able
zebra
Engstrцm
```

При упорядочении словаря приходится учитывать регистр букв, но совсем необязательно ударение. Для англоязычного пользователя приведенный выше перечень слов должен быть упорядочен следующим образом:

```
able
Ångström
Athens
zebra
Zulu
```

Но такой порядок следования слов неприемлем для шведскоязычного пользователя. Ведь в шведском языке буква **Å** отличается от буквы **A**, и поэтому она сортируется

после буквы **Z**! Это означает, что для шведскоязычного пользователя упомянутый выше перечень слов должен быть отсортирован следующим образом:

```
able
Athens
zebra
Zulu
Ångström
```

Чтобы получить компаратор с учетом региональных настроек, следует вызвать метод `Collator.getInstance()`:

```
Collator coll = Collator.getInstance(locale);
words.sort(coll);
    // Класс Collator реализует интерфейс Comparator<Object>
```

Для средств сортировки имеется пара дополнительных установок. В частности, можно установить *интенсивность* сортировки, чтобы откорректировать ее избирательность. Отличия в символах классифицируются как первичные, вторичные или третичные. Например, в английском языке отличие букв **e** и **f** считается первичным, отличие букв **e** и **é** — вторичным, а букв **e** и **E** — третичным.

Например, при обработке названий городов можно не принимать во внимание следующие отличия:

```
San José
San Jose
SAN JOSE
```

В этом случае средство сортировки настраивается в результате вызова следующего метода:

```
coll.setStrength(Collator.PRIMARY);
```

Более формальной оказывается установка *режима разложения на составляющие*, связанного с тем, что символ или последовательность символов можно иногда описывать в Юникоде разными способами. Например, буква **é** (**U+00E9**) может быть также выражена как обычная буква **e** (**U+0065**) и объединяющий знак ударения ´ (**U+0301**). Обычно эти отличия не имеют значения, но если они все же важны, то средство сортировки придется настроить следующим образом:

```
coll.setStrength(Collator.IDENTICAL);
coll.setDecomposition(Collator.NO_DECOMPOSITION);
```

С другой стороны, если требуется проявить нисхождение, рассматривая знак торговой марки ™ (**U+2122**) как сочетание букв **T** и **M**, то следует установить режим разложения на составляющие `Collator.FULL_DECOMPOSITION`.

Символьные строки, возможно, придется привести к нормализованным формам даже в том случае, если они не сортируются (например, для постоянного хранения или обмена с другой программой). В стандарте на Юникод определены четыре формы нормализации **C**, **D**, **KC** и **KD** (подробнее об этом — по адресу `www.unicode.`

org/unicode/reports/tr15/). В форме нормализации **C** символы с ударением всегда составляются. Например, буква **e** и объединяющий знак ударения ´ совместно образуют единый символ **é**. В форме нормализации **D** символы с ударением всегда разлагаются на составляющие их буквы и знаки ударения. Например, символ **é** превращается в последовательность, состоящую из буквы **e** и объединяющего знака ударения ´. В формах нормализации **KC** и **KD** символы также разлагаются на составляющие, как в приведенном выше примере со знаком торговой марки ™. Консорциум W3C рекомендует пользоваться формой нормализации **C** для передачи данных через Интернет.

Статический метод `normalize()` из класса `java.text.Normalizer` выполняет весь процесс нормализации. Ниже приведен характерный тому пример.

```
String city = "San Jose\u0301";
String normalized = Normalizer.normalize(city, Normalizer.Form.NFC);
```

13.6. Форматирование сообщений

При интернационализации прикладной программы нередко встречаются сообщения с переменными частями. Статический метод `format()` из класса `MessageFormat` принимает в качестве параметров строку шаблона с метками-заполнителями и значениями для меток-заполнителей следующим образом:

```
String template = "{0} has {1} messages"
String message = MessageFormat.format(template, "Pierre", 42);
```

Разумеется, вместо жесткого кодирования шаблон следует искать с учетом региональных настроек, например, `"Il y a {1} messages pour {0}"` на французском языке. О том, как это делается, речь пойдет в разделе 13.7.

Следует иметь в виду, что упорядочение меток-заполнителей может отличаться на разных языках. Так, сообщение `"Для Пьера имеются 42 сообщения"` на английском языке имеет вид `"Pierre has 42 messages"`, а на французском — `"Il y a 42 messages pour Pierre"`. При вызове метода `format()` метка-заполнитель **{0}** указывается в качестве аргумента сразу же после шаблона, а в качестве следующего аргумента — метка-заполнитель **{1}** и т.д.

Числа можно отформатировать как денежные суммы, добавив суффикс **number**, **currency** к метке-заполнителю следующим образом:

```
template="Your current total is {0,number,currency}."
```

В США числовое значение **1023,95** должно быть отформатировано в виде денежной суммы как **$1,023.95**. Это же значение в Германии должно быть представлено в виде денежной суммы как **1.023,95€** с учетом местной валюты и соблюдением правил, касающихся расположения разделителя десятичных цифр.

После признака **number** в упомянутом выше суффиксе может следовать признак **currency**, **integer**, **percent** или шаблон форматирования чисел из класса `DecimalFormat`, например **$,##0**. Значения дат из унаследованного класса `java.util.Date` можно отформатировать с помощью признака **date** или **time** и последующего

стиля форматирования **short**, **medium**, **long**, **full** или шаблона форматирования из класса SimpleDateFormat, например **yyyy-MM-dd**.

Следует иметь в виду, что значения объектов из пакета java.time требуют дополнительного преобразования, как показано в следующем примере кода:

```
String message =
  MessageFormat("It is now {0,time,short}.", Date.from(Instant.now()));
```

И наконец, средство форматирования choice позволяет формировать сообщения вроде следующего в зависимости от значения метки-заполнителя:

```
No files copied
1 file copied
42 files copied
```

Формат выбора (choice) состоит из последовательности пар, каждая из которых содержит нижний предел и строку форматирования. При этом нижний предел и строка форматирования разделяются знаком #, а пары — знаком |, как показано ниже.

```
String template = "{0,choice,0#No files|1#1 file|2#{0} files} copied";
```

Обратите внимание на то, что метка-заполнитель **{0}** дважды встречается в шаблоне. Когда формат выбора применяется к метке-заполнителю **{0}** со значением **42** в шаблоне форматирования сообщения, формат выбора возвращает символьную строку "{0} files". Эта строка снова форматируется, а полученный результат вставляется в сообщение.

НА ЗАМЕТКУ. Конструкция формата выбора несколько запутана. Так, если имеются три строки форматирования, то для их разделения требуются *два* предела. (В общем, пределов должно быть на единицу меньше, чем строк форматирования.) Но в классе **MessageFormat** первый предел на самом деле игнорируется!

Вместо знака # лучше употребить знак <, чтобы обозначить, что выбор должен быть сделан, если нижний предел строго меньше указанного значения. В качестве первого значения вместо знака # можно также употребить эквивалентный знак ≤ (**U+2264**) и указать нижний предел **-∞** (т.е. знак "минус" и знак бесконечности с кодом **U+221E**). Благодаря этому повышается удобочитаемость строки форматирования, как показано ниже.

```
-∞<No files|0<1 file|2≤{0}files
```

ВНИМАНИЕ. Любой текст в одиночных кавычках ' . . . ' включается в шаблон буквально. Например, строка '**{0}**' обозначает *не* метку-заполнитель, а просто текст **{0}**. Если же в шаблоне требуются одиночные кавычки, они должны быть указаны *дважды*, как показано ниже.

```
String template = "<a href=''{0}''>{1}</a>";
```

В статическом методе `MessageFormat.format()` применяются текущие региональные настройки для форматирования значений. Если же потребуется отформатировать значения с произвольными региональными настройками, то сделать это будет немного труднее, поскольку для подобной цели не предоставляется метод с аргументами переменной длины. В таком случае форматируемые значения придется разместить в массиве типа `Object[]` следующим образом:

```
MessageFormat mf = new MessageFormat(template, locale);
String message = mf.format(new Object[] { arg1, arg2, ... });
```

13.7. Комплекты ресурсов

При локализации прикладной программы ее исходный код лучше всего отделить от символьных строк сообщений, меток экранных кнопок и прочего текста, который требуется перевести на другие языки. В языке Java весь этот текст можно разместить в так называемых *комплектах ресурсов*, а затем передать эти комплекты переводчикам, которые могут отредактировать их, не затрагивая исходный код программы.

НА ЗАМЕТКУ. В главе 4 описывается понятие архивного JAR-файла ресурсов, где могут быть размещены файлы данных, фонограмм и изображений. С помощью метода **`getResource()`** из класса **`Class`** можно найти этот файл, открыть его и получить URL нужного ресурса. Это удобный механизм для комплектации прикладной программы файлами ресурсов, но он не поддерживает региональные настройки.

13.7.1. Организация комплектов ресурсов

Для локализации прикладной программы обычно создается ряд комплектов ресурсов. Каждый такой комплект представляет собой файл свойств или специальный класс ресурсов с записями или полями для отдельных региональных настроек или ряда совпадающих региональных настроек.

В этом разделе обсуждаются только файлы свойств, поскольку они применяются намного чаще, чем классы ресурсов. Файл свойств представляет собой текстовый файл с расширением **`.properties`**, содержащий пары "ключ–значение". Например, файл свойств `messages_de_DE.properties` может содержать следующее:

```
computeButton=Rechnen
cancelButton=Abbrechen
defaultPaperSize=A4
```

Для файлов, образующих подобные комплекты ресурсов, требуется выработать специальные условные обозначения. Например, ресурсы, характерные для Германии, должны храниться в файле ***имя_комплекта*`_de_DE`**, тогда как ресурсы, общие для всех немецкоязычных стран, — в файле ***имя_комплекта*`_de`**. Ниже перечислены условные обозначения имен файлов, которые могут быть *претендентами* на комплект ресурсов для отдельного сочетания языка, письма и страны.

имя_комплекта_язык_письмо_страна
имя_комплекта_язык_письмо
имя_комплекта_язык_страна
имя_комплекта_язык

Если условное обозначение ***имя_комплекта*** содержит точки, то файл может быть размещен в соответствующем каталоге. Например, файлы для комплекта ресурсов `com.mycompany.messages` размещаются в каталоге `com/mycompany/messages_de_DE.properties` и т.д.

Чтобы загрузить комплект ресурсов для региональных настроек по умолчанию, достаточно сделать следующий вызов:

```
ResourceBundle res = ResourceBundle.getBundle(имя_комплекта);
```

а для заданных региональных настроек такой вызов:

```
ResourceBundle bundle = ResourceBundle.getBundle(имя_комплекта, locale);
```

ВНИМАНИЕ. При первом из приведенных выше вызовов метода **`getBundle()`** применяются общие региональные настройки по умолчанию, а *не только* для отображения. Если же требуется найти ресурс для отображения пользовательского интерфейса, то в качестве региональных настроек этому методу следует передать результат вызова **`Locale.getDisplayDefault()`**.

Чтобы найти символьную строку в комплекте ресурсов по заданному ключу, достаточно вызвать метод `getString()` следующим образом:

```
String computeButtonLabel = bundle.getString("computeButton");
```

Правила загрузки комплектов ресурсов непросты и делятся на две стадии. На первой стадии обнаруживается соответствующий комплект ресурсов. Этот процесс происходит в следующие три этапа.

1. Сначала ищутся все сочетания имени комплекта ресурсов, языка, письма, страны и варианты в приведенном выше порядке до тех пор, пока не будет обнаружено совпадение. Так, если целевыми являются региональные настройки **de-DE**, а файл свойств `messages_de_DE.properties` отсутствует, но имеется файл свойств `messages_de.properties`, то именно он и становится совпавшим комплектом ресурсов.
2. Если совпадение не обнаружено, то данный процесс повторяется с региональными настройками по умолчанию. Так, если требуется немецкий комплект ресурсов, но таковой отсутствует, то по умолчанию используются региональные настройки **en-US**, а в качестве совпавшего комплекта ресурсов выбирается файл свойств `messages_en_US.properties`.
3. Если же совпадение не обнаружено и с региональными настройками по умолчанию, то в качестве совпавшего комплекта ресурсов выбирается файл свойств `messages.properties` (т.е. без всяких суффиксов). А если и он отсутствует, то поиск приводит к неудачном исходу.

НА ЗАМЕТКУ. Для вариантов упрощенного и традиционного китайского письма и разновидностей норвежского языка имеются особые правила. Подробнее об этом см. в документирующих комментариях к классу **`ResourceBundle.Control`**.

На второй стадии обнаруживаются *родители* совпадающего комплекта ресурсов. Родителями являются файлы свойств, находящиеся ниже совпадающего комплекта ресурсов в списке претендентов, а также файл свойств без всяких суффиксов. Например, родителями комплекта ресурсов `messages_de_DE.properties` являются файлы свойств `messages_de.properties` и `messages.properties`. Поиск по ключам в совпадающем комплекте ресурсов и его родителях осуществляет методом `getString()`.

НА ЗАМЕТКУ. Если совпадающий комплект ресурсов найден на первой стадии, то его родители вообще не выбираются из региональных настроек по умолчанию.

ВНИМАНИЕ. Содержимое файлов свойств представлено в коде ASCII. Все символы вне кода ASCII должны быть непременно представлены в кодировке **`\uxxxx`**. Например, значение **`Préférences`** указывается следующим образом:

```
preferences=Pr\u00E9fer\u00E9nces
```

Для формирования этих файлов можно воспользоваться инструментальным средством **`native2ascii`**.

13.7.2. Классы комплектов ресурсов

Чтобы предоставить ресурсы не в виде символьных строк, следует определить классы, расширяющие класс `ResourceBundle`. Для этой цели выбираются такие же условные обозначения, как и для файлов свойств. Ниже приведены характерные тому примеры.

```
com.mycompany.MyAppResources_en_US
com.mycompany.MyAppResources_de
com.mycompany.MyAppResources
```

Чтобы реализовать класс ресурсов, достаточно расширить класс `ListResourceBundle`. Все ресурсы размещаются в массиве пар "ключ–значение" и возвращаются из метода `getContents()`, как показано в следующем примере кода:

```
package com.mycompany;
public class MyAppResources_de extends ListResourceBundle {
  public Object[][] getContents() {
    return new Object[][] {
      { "backgroundColor", Color.BLACK },
      { "defaultPaperSize", new double[] { 210, 297 } }
    };
  }
}
```

Чтобы получить объекты из такого комплекта ресурсов, следует вызвать метод `getObject()`, как показано ниже.

```
ResourceBundle bundle =
    ResourceBundle.getBundle("com.mycompany.MyAppResources", locale);
Color backgroundColor = (Color) bundle.getObject("backgroundColor");
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

ВНИМАНИЕ. В методе **`ResourceBundle.getBundle()`** отдается предпочтение классам ресурсов над файлами свойств, когда в нем обнаруживается класс ресурсов и файл свойств с одним и тем же именем комплекта ресурсов.

13.8. Кодировки символов

Тот факт, что в Java применяется Юникод, совсем не означает, что все затруднения с кодировками символов устранены. Правда, беспокоиться о кодировке символов в объектах типа `String` не приходится. Любая получаемая символьная строка, будь то аргумент командной строки, данные, введенные с консоли или в текстовом поле ГПИ, будут представлены в виде строки символов в кодировке UTF-16, составляющих введенный пользователем текст.

При отображении символьной строки виртуальная машина кодирует ее для локальной платформы. И здесь возможны два затруднения. Вполне возможно, что в шрифте для отображения отсутствует символический знак (так называемый глиф) для конкретного символа в Юникоде. В ГПИ, простроенном на Java, такие символы отображаются полыми прямоугольниками. Для вывода на консоль отсутствующие символы отображаются знаком ?, если на консоли применяется кодировка символов, которая не способна представить все выводимые символы. Пользователи могут разрешить подобные затруднения, установив соответствующие шрифты или переключив консоль на кодировку UTF-8.

Дело усложняется еще больше, когда прикладная программа читает файлы открытого текста, создаваемые пользователями. Простые текстовые редакторы нередко производят файлы в кодировке локальной платформы. Эту кодировку можно получить, сделав следующий вызов:

```
Charset platformEncoding = Charset.defaultCharset();
```

Это вполне обоснованное предположение относительно предпочитаемой пользователем кодировки символов. Но пользователю должна быть предоставлена возможность изменить ее.

Если требуется предоставить выбор кодировок символов, их локализованные наименования можно получить следующим образом:

```
String displayName = encoding.displayName(locale);
    // Получает наименования кодировок вроде UTF-8,
    // ISO-8859-6 или GB18030
```

К сожалению, эти наименования на самом деле непригодны для конечных пользователей. Ведь им приходится выбирать между Юникодом, арабской, упрощенной китайской и прочей кодировкой символов.

СОВЕТ. Исходные файлы на языке Java также являются текстовыми файлами. Если вы не участвуете в проекте как программист, не сохраняйте исходные файлы в кодировке, принятой на данной платформе. Любые символы или комментарии, которые нельзя представить в коде ASCII, можно представить в виде управляющих последовательностей `\uxxxx`, но это трудоемкий процесс. Вместо этого лучше воспользоваться кодировкой UTF-8. С этой целью настройте глобальные параметры текстового редактора и консоли на кодировку UTF-8 или выполните компиляцию по следующей команде:

```
javac -encoding UTF-8 *.java
```

13.9. Глобальные параметры настройки

В завершение этой главы рассмотрим прикладной программный интерфейс API, который косвенно связан с интернационализацией. Он предназначен для хранения пользовательских глобальных параметров, к которым могут быть отнесены и предпочитаемые региональные настройки. Безусловно, глобальные параметры можно хранить в файле свойств, загружаемом при запуске программы на выполнение. Но для именования и размещения файлов конфигурации отсутствуют стандартные соглашения, что повышает вероятность конфликтов при установке пользователями многих прикладных программ на Java.

В некоторых операционных системах имеется центральное хранилище данных конфигурации. Характерным тому примером служит системный реестр в Microsoft Windows. В классе `Preferences`, реализующем в Java стандартный механизм сохранения пользовательских глобальных параметров, для этой цели применяется системный реестр Windows. А в Linux данные конфигурации хранятся в локальной файловой системе. Конкретная реализация хранилища доступна для программистов через класс `Preferences`.

В хранилище типа `Preferences` хранится дерево узлов. В каждом его узле находится таблица, содержащая пары "ключ–значение". Значения могут быть числовыми, логическими типа `boolean`, символьными строками или массивами байтов.

НА ЗАМЕТКУ. Для сохранения произвольных объектов ничего не предусмотрено. Безусловно, объекты можно подвергать сериализации и сохранять в виде массива байтов, если речь не идет о длительном хранении.

Пути к узлам выглядят как `/com/mycompany/myapp`. Как и имена пакетов, пути к узлам можно начинать в обратном порядке с имен доменов, чтобы избежать конфликтов имен.

Имеются два параллельных дерева. Каждому пользователю прикладной программы доступно одно дерево, а другое, называемое системным, предназначено для хранения настроек, общих для всех пользователей. Для доступа к соответствующему

пользовательскому дереву в классе Preferences употребляется принятое в операционной системе обозначение "текущий пользователь". Доступ к узлу дерева начинается с пользовательского или системного корневого узла одним из следующих способов:

```
Preferences root = Preferences.userRoot();
```

или

```
Preferences root = Preferences.systemRoot();
```

После этого доступ к отдельным узлам осуществляется по путям к ним следующим образом:

```
Preferences node = root.node("/com/mycompany/myapp");
```

С другой стороны, статическому методу userNodeForPackage() или systemNodeForPackage() в качестве параметров можно предоставить объект типа Class, а также путь к узлу, производный от имени пакета данного класса:

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

Имея в своем распоряжении узел, можно получить доступ к таблице с парами "ключ–значение". Символьная строка из этой таблицы извлекается следующим образом:

```
String preferredLocale = node.get("locale", "");
```

А для остальных типов данных служат следующие методы:

```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

При чтении информации нужно указать значение по умолчанию на тот случай, если хранилище недоступно. С другой стороны, данные можно записать в хранилище с помощью метода put() следующим образом:

```
void put(String key, String value)
void putInt(String key, int value)
```

и так далее. Чтобы удалить запись из узла, достаточно вызвать следующий метод:

```
void remove(String key)
```

А для удаления всего узла и его потомков следует сделать вызов node.removeNode(). Все ключи, хранящиеся в узле и пути ко всем его потомкам, можно перечислить с помощью следующих методов:

```
String[] keys()
String[] childrenNames()
```

НА ЗАМЕТКУ. Выяснить тип значения по конкретному ключу никак нельзя.

Экспортировать глобальные параметры в поддерево можно, вызвав следующий метод для корневого узла этого поддерева:

```
void exportSubtree(OutputStream out)
```

Данные сохраняются в формате XML. Их можно импортировать в другое хранилище следующим образом:

```
InputStream in = Files.newInputStream(path)
Preferences.importPreferences(in);
```

Упражнения

1. Напишите программу, демонстрирующую стили форматирования даты и времени во французском, китайском и тайском представлении.
2. В каких из региональных настроек вашей виртуальной машины JVM не употребляются арабские цифры для форматирования чисел?
3. В каких из региональных настроек вашей виртуальной машины JVM применяются те же самые условные обозначения дат (в формате месяц/день/год), что и в США?
4. Напишите программу, выводящую названия всех языков из региональных настроек вашей виртуальной машины JVM на всех имеющихся языках. Отсортируйте их, исключив дубликаты.
5. Повторите предыдущее упражнение для наименований денежных единиц.
6. Напишите программу, выводящую списком все денежные единицы, обозначаемые разными знаками хотя бы в двух региональных настройках.
7. Напишите программу, выводящую списком отображаемые и самостоятельные названия месяцев во всех региональных настройках, где они отличаются, учитывая то, что самостоятельные названия месяцев могут состоять из цифр.
8. Напишите программу, выводящую списком все символы в Юникоде, расширяемые до двух и больше символов в коде ASCII в форме нормализации `KC` или `KD`.
9. Попробуйте осуществить интернационализацию всех сообщений в одной из ваших прикладных программ хотя бы на двух языках, используя комплекты ресурсов.

10. Предоставьте механизм для представления имеющихся кодировок символов в удобочитаемом виде, как в вашем веб-браузере. Названия языков должны быть локализованы. (Воспользуйтесь переводами на местные языки.)
11. Предоставьте класс для отображения форматов бумаги с учетом региональных настроек, используя предпочтительные единицы измерения и стандартный формат бумаги в заданных региональных настройках. (Во всем мире, кроме США и Канады, употребляются форматы бумаги по стандарту ISO 216. И только в трех странах официально не принята метрическая система: Либерии, Бирме и США.)

Глава 14

Компиляция и написание сценариев

В этой главе...

- 14.1. Прикладной программный интерфейс API компилятора
- 14.2. Прикладной программный интерфейс API для сценариев
- 14.3. Интерпретатор Nashorn
- 14.4. Написание сценариев командного процессора средствами Nashorn
- Упражнения

В этой главе поясняется, как пользоваться прикладным программным интерфейсом API компилятора, чтобы компилировать исходный код, написанный на Java, из другой прикладной программы. В ней также будет показано, как выполнять программы, написанные на других языках, из программ на Java, используя прикладной программный интерфейс API для сценариев. Это особенно удобно, если пользователям требуется предоставить возможность расширить прикладную программу сценариями.

Основные положения этой главы приведены ниже.

1. С помощью прикладного программного интерфейса API компилятора можно динамически генерировать код Java и компилировать его.
2. Прикладной программный интерфейс API для сценариев дает возможность организовать взаимодействие программы на Java со сценариями на других языках.
3. В комплект JDK входит интерпретатор Nashorn языка JavaScript, обладающий хорошей производительностью и соответствующий стандарту JavaScript.
4. Интерпретатор Nashorn предоставляет удобный синтаксис для обращения со списками и отображениями в Java, а также со свойствами компонентов JavaBeans.
5. В интерпретаторе Nashorn поддерживаются лямбда-выражения и ограниченный механизм для расширения классов и реализации интерфейсов в Java.
6. В интерпретаторе Nashorn поддерживается написание сценариев командного процессора на JavaScript.

14.1. Прикладной программный интерфейс API компилятора

Имеется немало инструментальных средств, где требуется скомпилировать код Java. Очевидно, что к их числу относятся среды разработки, программы, обучающие программированию на Java, а также инструментальные средства тестирования и автоматизации сборки программ. Еще одним примером может служить обработка веб-страниц типа JavaServer Pages со встроенными операторами Java.

14.1.1. Вызов компилятора

Вызвать компилятор Java очень просто. Ниже приведен характерный тому пример. Нулевое значение переменной `result` указывает на успешный исход компиляции.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ...;
OutputStream errStream = ...;
int result = compiler.run(null, outStream, errStream,
    "-sourcepath", "src", "Test.java");
```

Компилятор направляет выводимый результат и сообщения об ошибках в предоставляемые потоки вывода. Если установить значение их параметров пустым

(null), то вместо них в методе run() будут использованы стандартные потоки вывода System.out и System.err. А в качестве первого параметра метода run() служит поток ввода. Компилятор не принимает данные, вводимые с консоли, и поэтому значение этого параметра может всегда оставаться пустым (null). Следует иметь в виду, что метод run() наследуется из обобщенного интерфейса Tool, что удобно для тех инструментальных средств, которые читают вводимые данные.

В качестве остальных параметров метода run() служат аргументы, которые должны быть переданы утилите **javac**, если вызывать ее из командной строки. Это могут быть параметры командной строки или имена файлов.

14.1.2. Запуск задания на компиляцию

Объект типа CompilationTask обеспечивает более полный контроль над процессом компиляции. Это может быть удобно в том случае, если требуется предоставить исходный код из символьной строки, зафиксировать файлы классов в оперативной памяти или обработать сообщения об ошибках и предупреждениях.

Чтобы получить объект типа CompilationTask, нужно начать с объекта compiler, как было показано в примере кода из предыдущего раздела, а затем сделать следующий вызов:

```
JavaCompiler.CompilationTask task = compiler.getTask(
    errorWriter, // если этот аргумент принимает пустое значение null,
        // то используется стандартный поток вывода ошибок System.err
    fileManager, // если этот аргумент принимает пустое значение null,
                 // то используется стандартный диспетчер файлов
    diagnostics, // если этот аргумент принимает пустое значение null,
        // то используется стандартный поток вывода ошибок System.err
    options, // если параметры командной строки отсутствуют, то
             // этот аргумент принимает пустое значение null
    classes, // этот аргумент служит для обработки аннотаций;
             // в ее отсутствие он принимает пустое значение null
    sources);
```

Три последних аргумента метода getTask() являются экземплярами Iterable типа String. Например, последовательность параметров командной строки может быть задана следующим образом:

```
Iterable<String> options = Arrays.asList("-d", "bin");
```

Аргумент sources принимает экземпляры Iterable типа JavaFileObject. Так, если требуется скомпилировать дисковые файлы, следует получить диспетчер файлов типа StandardJavaFileManager и вызвать его метод getJavaFileObjects(), как показано ниже.

```
StandardJavaFileManager fileManager =
    compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> sources =
    fileManager.getJavaFileObjectsFromFiles("File1.java", "File2.java");
JavaCompiler.CompilationTask task = compiler.getTask(
    null, null, null, options, null, sources);
```

НА ЗАМЕТКУ. Аргумент `classes` служит только для обработки аннотаций. В этом случае нужно также вызвать метод `task.processors(annotationProcessors)` со списком объектов типа `Processor`. Пример обработки аннотаций см. в главе 11.

Метод `getTask()` возвращает объект задания, но еще не начинает процесс компиляции. Класс `CompilationTask` расширяет интерфейс `Callable<Boolean>`. Его объект можно передать интерфейсу `ExecutorService` для параллельного исполнения или просто сделать синхронный вызов следующим образом:

```
Boolean success = task.call();
```

14.1.3. Чтение исходных файлов из оперативной памяти

Если сгенерировать исходный код динамически, то его можно скомпилировать из оперативной памяти, не сохраняя файлы на диске. Для хранения полученного кода служит следующий класс:

```
public class StringSource extends SimpleJavaFileObject {
  private String code;

  StringSource(String name, String code) {
    super(URI.create("string:///" + name.replace('.','/') + ".java"),
          Kind.SOURCE);
    this.code = code;
  }

  public CharSequence getCharContent(boolean ignoreEncodingErrors) {
    return code;
  }
}
```

Это дает возможность сгенерировать код для разрабатываемых классов и предоставить компилятору список объектов типа `StringSource`, как показано ниже.

```
String pointCode = ...;
String rectangleCode = ...;
List<StringSource> sources = Arrays.asList(
    new StringSource("Point", pointCode),
    new StringSource("Rectangle", rectangleCode));
task = compiler.getTask(null, null, null, null, null, sources);
```

14.1.4. Запись байт-кодов в оперативную память

Если скомпилировать классы динамически, то сохранять файлы на диске не потребуется. Их можно сохранить в оперативной памяти, откуда загружать их непосредственно.

Прежде всего, для хранения байтов в оперативной памяти служит следующий класс:

```
public class ByteArrayClass extends SimpleJavaFileObject {
  private ByteArrayOutputStream out;
```

```
  ByteArrayClass(String name) {
    super(URI.create("bytes:///" + name.replace('.','/') + ".class"),
          Kind.CLASS);
  }

  public byte[] getCode() {
    return out.toByteArray();
  }

  public OutputStream openOutputStream() throws IOException {
    out = new ByteArrayOutputStream();
    return out;
  }
}
```

Затем нужно предоставить диспетчер файлов, чтобы воспользоваться этими классами для вывода, как показано ниже.

```
List<ByteArrayClass> classes = new ArrayList<>();
StandardJavaFileManager stdFileManager =
    compiler.getStandardFileManager(null, null, null);
JavaFileManager fileManager =
    new ForwardingJavaFileManager<JavaFileManager>(stdFileManager) {
      public JavaFileObject getJavaFileForOutput(Location location,
          String className, Kind kind, FileObject sibling)
              throws IOException {
          if (kind == Kind.CLASS) {
              ByteArrayClass outfile = new ByteArrayClass(className);
              classes.add(outfile);
              return outfile;
          } else
            return super.getJavaFileForOutput(
            location, className, kind, sibling);
          }
      };
```

Для загрузки классов потребуется следующий их загрузчик (см. главу 4):

```
public class ByteArrayClassLoader extends ClassLoader {
  private Iterable<ByteArrayClass> classes;

  public ByteArrayClassLoader(Iterable<ByteArrayClass> classes) {
     this.classes = classes;
  }

  @Override public Class<?> findClass(String name)
            throws ClassNotFoundException {
    for (ByteArrayClass cl : classes) {
      if (cl.getName().equals("/" + name.replace('.','/') + ".class")) {
        byte[] bytes = cl.getCode();
        return defineClass(name, bytes, 0, bytes.length);
      }
    }
    throw new ClassNotFoundException(name);
  }
}
```

По завершении компиляции вызывается метод `Class.forName()` с данным загрузчиком классов, как показано ниже.

```
ByteArrayClassLoader loader = new ByteArrayClassLoader(classes);
Class<?> cl = Class.forName("Rectangle", true, loader);
```

14.1.5. Фиксация диагностической информации

Для приема сообщений об ошибках устанавливается диагностический приемник типа `DiagnosticListener`, который получает объект типа `Diagnostic` всякий раз, когда компилятор посылает сообщение об ошибке или предупреждении. Интерфейс `DiagnosticListener` реализуется классом `DiagnosticCollector`. Он просто накапливает всю диагностическую информацию для ее обработки по завершении компиляции:

```
DiagnosticCollector<JavaFileObject> collector = new DiagnosticCollector<>();
compiler.getTask(null, fileManager, collector, null, null, sources).call();
for (Diagnostic<? extends JavaFileObject> d : collector.getDiagnostics()) {
  System.out.println(d);
}
```

Объект типа `Diagnostic` содержит информацию о месте неполадки, включая имя файла, номер строки и столбца, а также удобочитаемое описание неполадки. Приемник типа `DiagnosticListener` можно также установить в стандартном диспетчере файлов на тот случай, если потребуется перехватывать сообщения об отсутствующих файлах, как показано ниже.

```
StandardJavaFileManager fileManager =
    compiler.getStandardFileManager(diagnostics, null, null);
```

14.2. Прикладной программный интерфейс API для сценариев

Для написания сценариев существует специальный язык, интерпретирующий исходный текст программы во время выполнения, минуя цикл редактирования, компиляции, компоновки и выполнения. Такой подход побуждает к экспериментированию. Кроме того, языки написания сценариев менее сложны, что делает их пригодными для расширения прикладных программ опытными пользователями.

Прикладной программный интерфейс API для сценариев дает возможность сочетать преимущества написания сценариев с традиционным программированием. Он позволяет вызывать из программы на Java сценарии, написанные на JavaScript, Groovy, Ruby и даже на таких экзотических языках, как Scheme и Haskell. В последующих разделах будет показано, как выбирать механизм для конкретного языка, выполнять сценарии и выгодно пользоваться расширенными возможностями, предоставляемыми некоторыми механизмами сценариев.

14.2.1. Получение механизма сценариев

Механизм сценариев представляет собой библиотеку, предназначенную для выполнения сценариев на конкретном языке. При запуске виртуальной машины

обнаруживаются доступные механизмы сценариев. Для их перечисления строится объект типа `ScriptEngineManager` и вызывается метод `getEngineFactories()`.

Как правило, нужный механизм сценариев известен заранее. Его можно запросить по имени, как показано в следующем примере кода:

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");
```

В состав комплекта разработки Java Development Kit входит интерпретатор сценариев на JavaScript, называемый Nashorn и описываемый далее, в разделе 14.3. Для поддержки сценариев на других языках можно предоставить архивные JAR-файлы по пути к классам. Теперь уже нет официального списка языков написания сценариев для интеграции с Java. Для этой цели достаточно выбрать наиболее предпочтительный механизм и найти поддержку нужного языка в Java по спецификации JSR 223.

Имея в своем распоряжении механизм сценариев, для обращения к сценарию достаточно сделать следующий вызов:

```
Object result = engine.eval(scriptString);
```

Сценарий можно также прочитать из потока чтения типа `Reader` следующим образом:

```
Object result = engine.eval(Files.newBufferedReader(path, charset));
```

Один и тот же механизм позволяет вызывать многие сценарии. Если в одном сценарии определяются переменные, функции или классы, то их определения сохраняются в большинстве механизмов сценариев для последующего применения. Например, в результате выполнения следующего фрагмента кода:

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

возвращается значение **1729**.

НА ЗАМЕТКУ. Чтобы выяснить, насколько надежным может быть параллельное выполнение сценариев в нескольких потоках, достаточно вызвать метод **`engine.getFactory().getParameter("THREADING")`**, возвращающий одно из следующих значений:

- **`null`** — параллельное выполнение сценариев ненадежно;
- **`"MULTITHREADED"`** — параллельное выполнение сценариев надежно. Результаты, получаемые в одном потоке исполнения, недоступны в другом потоке;
- **`"THREAD-ISOLATED"`** — для каждого потока исполнения сохраняются разные привязки переменных;
- **`"STATELESS"`** — привязки переменных не изменяются в сценариях.

14.2.2. Привязки

Привязка состоит из имени переменной и связанного с ней объекта Java. В качестве примера рассмотрим следующие операторы:

```
engine.put("k", 1728);
Object result = engine.eval("k + 1");
```

С другой стороны, можно извлечь переменные, привязанные операторами сценария, следующим образом:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

Эти привязки существуют в области *действия механизма сценариев*. Кроме того, имеется глобальная область действия. Любые привязки, вводимые в объект типа `ScriptEngineManager`, доступны для всех механизмов сценариев.

Вместо ввода привязок в механизм сценариев или глобальную область действия их можно накапливать в объекте типа `Bindings` и передавать методу `eval()`, как показано ниже. Это удобно в том случае, если ряд привязок не должен существовать для последующих вызовов метода `eval()`.

```
Bindings scope = engine.createBindings();
scope.put("k", 1728);
Object result = engine.eval("k + 1", scope);
```

14.2.3. Переадресация ввода-вывода

Стандартный ввод-вывод из сценариев переадресуется путем вызова методов `setReader()` и `setWriter()` в контексте сценария. Любой вывод, направляемый функцией `print()` языка JavaScript, посылается объекту `writer`, как показано в приведенном ниже примере кода.

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(writer);
engine.eval("print('Hello')");
String result = writer.toString();
```

Методы `setReader()` и `setWriter()` оказывают влияние только на источники ввода-вывода из механизма сценариев. Так, если выполнить следующий код на JavaScript:

```
print('Hello');
java.lang.System.out.println('World');
```

то переадресуется только первый вывод.

НА ЗАМЕТКУ. В интерпретаторе Nashorn отсутствует понятие стандартного источника ввода. В частности, вызов метода **`setReader()`** не оказывает никакого влияния.

НА ЗАМЕТКУ. В JavaScript точки с запятой в конце строки кода указывать необязательно. И хотя многие программирующие на JavaScript все равно их указывают, в примерах кода из этой главы они опускаются, чтобы было легче различать фрагменты кода Java и JavaScript. По этой же причине в коде JavaScript символьные строки заключаются там, где это возможно, в одиночные кавычки `'...'`, а не в двойные `"..."`.

14.2.4. Вызов функций и методов из сценариев

С помощью некоторых механизмов сценариев можно вызывать функции на языке написания сценариев, не вычисляя код для вызова в качестве сценария. Это удобно в том случае, если пользователям требуется предоставить возможность реализовать службу на избранном ими языке написания сценариев, чтобы обращаться к ней из программы на Java.

С этой целью механизмы написания сценариев, предоставляющие подобные возможности, в том числе интерпретатор Nashorn, реализуют интерфейс `Invocable`. Чтобы вызвать функцию из сценария, достаточно вызвать метод `invokeFunction()`, передав ему имя функции и ее аргументы, как показано в следующем примере кода:

```
// определить в JavaScript функцию для вывода приветствия
engine.eval("function greet(how, whom)
           { return how + ', ' + whom + '!' }");

// вызвать функцию с аргументами "Hello", "World"
result = ((Invocable) engine).invokeFunction("greet", "Hello", "World");
```

Если язык написания сценариев является объектно-ориентированным, то можно вызвать метод `invokeMethod()`:

```
// определить в JavaScript класс Greeter
engine.eval("function Greeter(how) { this.how = how }");
engine.eval("Greeter.prototype.welcome =
     " + " function(whom) { return this.how + ', ' + whom + '!' }");
// получить экземпляр этого класса
Object yo = engine.eval("new Greeter('Yo')");

// вызвать метод welcome() для экземпляра этого класса
result = ((Invocable) engine).invokeMethod(yo, "welcome", "World");
```

НА ЗАМЕТКУ. Подробнее о том, каким образом классы определяются в JavaScript, можно узнать из книги *JavaScript—The Good Parts* Дугласа Крокфорда (Douglas Crockford, издательство O'Reilly, 2008 г.)

НА ЗАМЕТКУ. Даже если интерфейс **`Invocable`** и не реализуется в сценарии, вызвать нужный метод независимо от конкретного языка все-таки можно. В частности, метод **`getMethodCallSyntax()`** из класса **`ScriptEngineFactory`** производит символьную строку, которую можно передать методу **`eval()`**.

В качестве следующего шага можно обратиться к механизму сценариев за реализацией интерфейса Java. Это даст возможность вызывать функции из сценария, используя синтаксис вызова методов Java.

Подробности этого процесса зависят от конкретного механизма сценариев, но, как правило, для каждого метода из интерфейса нужно предоставить соответствующую функцию. В качестве примера рассмотрим следующий интерфейс Java:

```
public interface Greeter {
  String welcome(String whom);
}
```

Если определить глобальную функцию с тем же самым именем **welcome** в интерпретаторе Nashorn, то ее можно будет вызвать через данный интерфейс:

```
// определить в JavaScript функцию welcome()
engine.eval("function welcome(whom) { return 'Hello, ' + whom + '!' }");
// получить объект Java и вызвать метод Java
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
result = g.welcome("World");
```

В объектно-ориентированном языке написания сценариев класс может быть доступен через соответствующий интерфейс Java. В качестве примера ниже показано, каким образом вызывается объект из класса `Greeter` в JavaScript с помощью синтаксиса Java. Более полезный пример приведен в упражнении 2 в конце этой главы.

```
Greeter g = ((Invocable) engine).getInterface(yo, Greeter.class);
result = g.welcome("World");
```

14.2.5. Компилирование сценария

В некоторых механизмах код сценариев можно скомпилировать в промежуточной форме для более эффективного его выполнения. С этой целью в подобных механизмах реализуется интерфейс `Compilable`. В приведенном ниже примере демонстрируется компиляция и вычисление кода из файла сценария. Разумеется, компилировать сценарий имеет смысл только в том случае, если в нем имеется немалый объем работы или если его требуется выполнять часто.

```
if (engine implements Compilable) {
  Reader reader = Files.newBufferedReader(path, charset);
  CompiledScript script = ((Compilable) engine).compile(reader);
  script.eval();
}
```

14.3. Интерпретатор Nashorn

В комплект разработки Java Development Kit входит интерпретатор сценариев Nashorn, действующий очень быстро и в высокой степени совместимый с версией стандарта ECMAScript на языке JavaScript. Интерпретатором Nashorn можно пользоваться таким же образом, как и любым другим механизмом сценариев, но у него имеются специальные средства для взаимодействия с Java.

14.3.1. Выполнение Nashorn из командной строки

В состав версии Java 8 входит утилита **jjs**, действующая в режиме командной строки. Запустив ее на выполнение, можно выполнять команды JavaScript, как показано ниже.

```
$ jjs
jjs> 'Hello, World'
Hello, World
```

В итоге получается цикл, называемый REPL (чтение–вычисление–вывод) в таких языках, как Lisp, Scala и т.д. Всякий раз, когда вводится выражение, выводится вычисляемое его значение:

```
jjs> 'Hello, World!'.length
13
```

В режиме командной строки можно определять функции и вызывать их следующим образом:

```
jjs> function factorial(n) { return n <= 1 ? 1 : n * factorial(n - 1) }
function factorial(n) { return n <= 1 ? 1 : n * factorial(n - 1) }
jjs> factorial(10)
3628800
```

СОВЕТ. При написании более сложных функций целесообразно размещать код сценария JavaScript в отдельном файле и загружать его в режиме работы утилиты **jjs** по следующей команде:

```
load('functions.js')
```

Аналогичным образом можно вызывать и методы Java, как показано ниже. Если затем ввести **contents** в командной строке, то появится содержимое указанной веб-страницы.

```
var url = new java.net.URL('http://horstmann.com')
var input = new java.util.Scanner(url.openStream())
input.useDelimiter('$')
var contents = input.next()
```

Оцените, насколько это удобно. Вам не нужно беспокоиться об исключениях, но вы можете экспериментировать со сценариями в динамическом режиме. Что касается приведенного выше примера кода, то я поначалу не был уверен, удастся ли мне прочитать содержимое веб-страницы полностью, установив разделитель **$**. Но я попробовал, и у меня получилось. Для этого мне не пришлось писать метод `public static void main()`, определять типы переменных `input` и `contents`, не нужно было компилировать код и тем более создавать проект в ИСР. Цикл REPL служит простым и удобным способом исследования прикладного программного интерфейса API для сценариев. Управлять кодом Java из сценария JavaScript удобно, хотя и не совсем привычно.

СОВЕТ. Пользоваться циклом REPL в JavaScript было бы еще удобнее, если бы в нем поддерживалось редактирование в режиме командной строки. В Linux, Unix и Mac OS X можно установить утилиту **rlwrap** и выполнить команду **rlwrap jjs**. Затем для получения предыдущих команд достаточно нажать клавишу <> и далее отредактировать их. С другой стороны, утилиту **jjs** можно выполнить в текстовом редакторе Emacs. С этой целью запустите редактор Emacs и нажмите комбинацию клавиш <Alt+x> или <Esc+x> для перехода в режим командной строки и введите команду

`jjs`. Затем введите, как обычно, выражение. Для перехода на предыдущую или следующую строку нажмите комбинацию клавиш <Alt+p> или <Alt+n> соответственно, а для перемещения в пределах текущей строки — клавишу <←> или <→>. Отредактируйте команду, а затем нажмите клавишу <Enter>, чтобы выполнить команду.

14.3.2. Вызов методов получения, установки и перегружаемых методов

Если в Nashorn имеются объекты Java, то для них можно вызвать методы. Допустим, что экземпляр класса `NumberFormat` получается в Java следующим образом:

```
var fmt = java.text.NumberFormat.getPercentInstance()
```

В таком случае для него можно вызвать метод таким образом:

```
fmt.setMinimumFractionDigits(2)
```

Но если это метод получения или установки, то можно поступить еще лучше, воспользовавшись синтаксисом доступа к свойствам, как показано ниже.

```
fmt.minimumFractionDigits = 2
```

Если выражение `fmt.minimumFractionDigits` присутствует в левой части операции присваивания `=`, то оно преобразуется в вызов метода `setMinimumFractionDigits()`, а иначе — в вызов метода `fmt.getMinimumFractionDigits()`. Для доступа к свойствам можно даже воспользоваться синтаксисом квадратных скобок следующим образом:

```
fmt['minimumFractionDigits'] = 2
```

Обратите внимание на то, что аргументом операции `[]` служит символьная строка. В данном контексте это приносит мало пользы, но вызов `fmt[str]` можно сделать и со строковой переменной, чтобы получить доступ к произвольным свойствам.

В языке JavaScript отсутствует понятие перегрузки методов. В этом языке допускается только один метод с одним и тем же именем, хотя он и может иметь любое количество параметров какого угодно типа. Поэтому интерпретатор Nashorn пытается выбрать подходящий метод Java, исходя из количества и типов аргументов.

Как правило, имеется только один метод Java, соответствующий предоставляемым параметрам. В противном случае подходящий метод можно выбрать, используя следующий не совсем обычный синтаксис:

```
list['remove(Object)'](1)
```

В данном случае указывается метод `remove(Object)`, удаляющий из списка объект `1` типа `Integer`. (Имеется также метод `remove(int)`, удаляющий объект на позиции `1`.)

14.3.3. Построение объектов Java

Если объекты требуется построить в сценарии JavaScript, а не организовывать их передачу из механизма сценариев, то нужно знать, как получать доступ к пакетам Java. Для этой цели имеются два механизма.

С одной стороны, имеются глобальные объекты java, javax, javafx, com, org и edu для получения объектов из пакета или класса посредством записи через точку, как показано ниже. Если требуется доступ к пакету, имя которого не начинается с одного или двух приведенных ниже идентификаторов, его можно обнаружить в объекте типа Package, например Package.ch.cern.

```
var javaNetPackage = java.net // Объект типа JavaPackage
var URL = java.net.URL // Класс JavaClass
```

С другой стороны, можно вызвать функцию Java.type() следующим образом:

```
var URL = Java.type('java.net.URL')
```

Это немного быстрее, чем доступ по ссылке java.net.URL, да и лучше для проверки ошибок. (Если сделать опечатку, введя имя **java.net.Url**, Nashorn интерпретирует его как имя пакета.) Но если требуется быстрота и качество обработки ошибок, то вряд ли для этой цели стоит пользоваться языком написания сценариев. Поэтому лучше придерживаться более краткой формы доступа к пакетам.

НА ЗАМЕТКУ. Как следует из документации на интерпретатор Nashorn, объекты классов должны быть определены в самом начале файла сценария аналогично размещению операторов импорта пакетов в исходном файле Java. Ниже приведен характерный тому пример.

```
var URL = Java.type('java.net.URL')
var JMath = Java.type('java.lang.Math')
    // исключает конфликт с объектом типа Math в JavaScript
```

Получив объект класса, можно вызвать один из статических методов следующим образом:

```
JMath.floorMod(-3, 10)
```

Чтобы построить объект, оператору new в JavaScript следует передать объект класса. Любые параметры конструктора передаются обычным образом:

```
var URL = java.net.URL
var url = new URL('http://horstmann.com')
```

Если же эффективность не имеет особого значения, то можно сделать следующий вызов:

```
var url = new java.net.URL('http://horstmann.com')
```

ВНИМАНИЕ. Если метод **Java.type()** применяется вместе с оператором **new**, то для этой цели потребуются дополнительные скобки, как показано ниже.

```
var url = new (Java.type('java.net.URL'))('http://horstmann.com')
```

Если нужно указать внутренний класс, это можно сделать посредством записи через точку следующим образом:

```
var entry = new java.util.AbstractMap.SimpleEntry('hello', 42)
```

С другой стороны, если воспользоваться методом `Java.type()`, то следует ввести знак **$**, как это делается в виртуальной машине JVM и показано ниже.

```
var Entry = Java.type('java.util.AbstractMap$SimpleEntry')
```

14.3.4. Символьные строки в JavaScript и Java

Символьные строки в Nashorn, безусловно, являются объектами. В качестве примера рассмотрим следующий вызов в сценарии:

```
'Hello'.slice(-2) // возвращает подстроку 'lo'
```

В данном примере вызывается метод `slice()`, доступный в JavaScript. А в Java такой метод отсутствует. Но и следующий вызов:

```
'Hello'.compareTo('World')
```

вполне пригоден, несмотря на то, что в JavaScript отсутствует метод `compareTo()`, где вместо него просто употребляется операция сравнения **<**.

В данном случае символьная строка JavaScript преобразуется в символьную строку Java. В общем, такое преобразование происходит всякий раз, когда символьная строка JavaScript передается в качестве параметра методу Java.

Следует также заметить, что *любой* объект JavaScript преобразуется в символьную строку, когда он передается методу Java в виде параметра типа `String`. Рассмотрим следующий пример кода:

```
var path = java.nio.file.Paths.get(/home/)
    // Объект JavaScript типа RegExp преобразуется
    // в объект Java типа String!
```

В данном примере в качестве параметра методу `Paths.get()` передается регулярное выражение `/home/`. Но методу `Paths.get()` требуется параметр типа `String`, и он его получает, несмотря на то, что в данном случае это не имеет никакого смысла. И винить в этом интерпретатор Nashorn не стоит. Ведь он следует обычному для JavaScript правилу преобразовывать в символьную строку все подряд, если предполагается символьная строка. Аналогичное преобразование происходит с числовыми и логическими значениями. Например, вызов `'Hello'.slice('-2')` совершенно верен. Символьная строка `'-2'` негласно преобразуется в число **-2**. Благодаря именно этим особенностям программирование на подобных динамически типизированных языках превращается в захватывающее приключение.

14.3.5. Числа

В языке JavaScript отсутствует явная поддержка целых чисел. Тип `Number` в JavaScript аналогичен типу `double` в Java. Если число передается коду Java, где ожидается числовое значение типа `int` или `long`, любая дробная часть этого числа негласно отбрасывается. Например, вызов `'Hello'.slice(-2.99)` аналогичен вызову `'Hello'.slice(-2)`.

Из соображений эффективности интерпретатор Nashorn поддерживает вычисления целочисленными, если это возможно, но это отличие, как правило, очевидно. Ниже приведен пример того, как оно неочевидно.

```
var price = 10
java.lang.String.format('Price: %.2f', price)
  // ОШИБКА: формат f не годится для значений типа java.lang.Integer!
```

Значение переменной `price` оказывается целочисленным и присваивается экземпляру типа `Object`, поскольку у метода `format()` имеется параметр переменной длины типа `Object...`. Следовательно, интерпретатор Nashorn производит числовое значение типа `java.lang.Integer`. В итоге выполнить метод `format()` не удастся, поскольку формат **f** предназначен для чисел с плавающей точкой. В данном случае можно выполнить преобразование к типу `java.lang.Double` вручную, вызвав функцию `Number()` следующим образом:

```
java.lang.String.format('Unit price: %.2f', Number(price))
```

14.3.6. Обращение с массивами

Чтобы построить массив Java, нужно сначала создать объект класса, как показано ниже.

```
var intArray = Java.type('int[]')
var StringArray = Java.type('java.lang.String[]')
```

Затем нужно вызвать оператор `new`, предоставив длину массива следующим образом:

```
var numbers = new intArray(10) // Примитивный массив типа int[]
var names = new StringArray(10) // Массив ссылок на объекты типа String
```

Далее можно воспользоваться обозначением массива с помощью квадратных скобок обычным образом:

```
numbers[0] = 42
print(numbers[0])
```

Длину массива можно получить в виде свойства `numbers.length`. Для перебора всех значений в массиве `names` служит следующий цикл:

```
for each (var elem in names)
 Сделать что-нибудь с элементом elem массива
```

Этот цикл равнозначен расширенному циклу `for` в Java. Если же требуются индексные значения, то лучше воспользоваться следующим циклом:

```
for (var i in names)
   Сделать что-нибудь с переменной i и элементом массива names[i]
```

ВНИМАНИЕ. Несмотря на то что приведенный выше цикл внешне похож на расширенный цикл **for** в Java, в нем перебираются индексные значения. Массивы в JavaScript могут быть разреженными. Допустим, что массив инициализируется в JavaScript следующим образом:

```
var names = []
names[0] = 'Fred'
names[2] = 'Barney'
```

В таком случае в цикле **for (var i in names) print(i)** выводятся значения 0 и 2.

Массивы в Java и JavaScript заметно отличаются. Если массив JavaScript предоставляется там, где предполагается массив Java, то интерпретатор Nashorn выполнит необходимое преобразование. Но иногда для этого требуется дополнительная помощь. Так, из заданного массива JavaScript можно получить эквивалентный массив Java, вызвав метод `Java.to()` следующим образом:

```
var javaNames = Java.to(names, StringArray) // Массив типа String[]
```

С другой стороны, вызвав метод `Java.from()`, можно преобразовать массив Java в эквивалентный массив JavaScript таким образом:

```
var jsNumbers = Java.from(numbers)
jsNumbers[-1] = 42
```

Для устранения неоднозначности в связи с перегрузкой придется воспользоваться методом `Java.to()`. Например, следующий вызов:

```
java.util.Arrays.toString([1, 2, 3])
```

дает неоднозначный результат, поскольку Nashorn не в состоянии решить, следует ли выполнять преобразование исходного массива в массив типа `int[]` или `Object[]`. В таком случае лучше сделать следующий вызов:

```
java.util.Arrays.toString(Java.to([1, 2, 3], Java.type('int[]')))
```

или просто

```
java.util.Arrays.toString(Java.to([1, 2, 3], 'int[]'))
```

14.3.7. Списки и отображения

В интерпретаторе Nashorn предоставляется "синтаксическое удобство" для списков и отображений Java. Так, для вызова методов `get()` и `set()` вместе с любым объектом типа `List` в Java можно воспользоваться операцией `[]`, как показано ниже.

```
var names = java.util.Arrays.asList('Fred', 'Wilma', 'Barney')
var first = names[0]
names[0] = 'Duke'
```

Операция `[]` пригодна и для отображений в Java, как демонстрируется в следующем примере:

```
var scores = new java.util.HashMap
scores['Fred'] = 10 // вызывается метод scores.put('Fred', 10)
```

Для перебора всех элементов отображения можно воспользоваться циклом `for each` в JavaScript следующим образом:

```
for (var key in scores) ...
for each (var value in scores) ...
```

Если же ключи и значения требуется обработать вместе, то для этого достаточно обойти множество записей таким образом:

```
for each (var e in scores.entrySet())
    Обработать ключ e.key и значение e.value
```

НА ЗАМЕТКУ. Цикл **`for each`** подходит для любого класса Java, реализующего интерфейс **`Iterable`**.

14.3.8. Лямбда-выражения

В JavaScript применяются анонимные функции, как показано ниже.

```
var square = function(x) { return x * x }
    // В правой части данного выражения указана анонимная функция
var result = square(2)
    // Операция () вызывает функцию
```

Синтаксически анонимная функция очень похожа на лямбда-выражение в Java, только вместо стрелки после списка параметров в объявлении анонимной функции указывается ключевое слово **`function`**.

Как и лямбда-выражение в Java, анонимную функцию можно использовать в качестве аргумента метода из функционального интерфейса в Java. Ниже приведен характерный тому пример.

```
java.util.Arrays.sort(words,
  function(a, b) { return java.lang.Integer.compare(a.length, b.length) })
  // сортировать массив, увеличивая его длину
```

В механизме поддерживается сокращенное определение функций, тело которых состоит из единственного выражения. В определении таких функций можно опустить фигурные скобки и ключевое слово `return`, как показано ниже. И в этом случае следует отметить сходство данной анонимной функции с лямбда-выражением `(a, b) -> Integer. compare(a.length, b.length)` в Java.

```
java.util.Arrays.sort(words,
    function(a, b) java.lang.Integer.compare(a.length, b.length))
```

НА ЗАМЕТКУ. Сокращенное обозначение анонимных функций, иначе называемое *замыканием выражения*, не является частью официального стандарта на язык JavaScript (ECMAScript 5.1), но в то же время поддерживается реализацией JavaScript в Mozilla.

14.3.9. Расширение классов и реализация интерфейсов в Java

Чтобы расширить класс или реализовать интерфейс в Java, следует воспользоваться функцией `Java.extend()`. С этой целью предоставляется объект суперкласса или интерфейса, а также объект JavaScript вместе с методами, которые требуется переопределить или реализовать.

В качестве примера ниже приведен код итератора, производящего бесконечную последовательность произвольных чисел. В данном примере переопределяются методы `next()` и `hasNext()`. Для каждого из этих методов предоставляется отдельная реализация в виде анонимной функции в JavaScript.

```
var RandomIterator = Java.extend(java.util.Iterator, {
    next: function() Math.random(),
    hasNext: function() true
}) // объект класса RandomIterator
var iter = new RandomIterator() // использовать этот объект для
                                // построения экземпляра
```

НА ЗАМЕТКУ. При вызове метода **`Java.extend()`** можно указать любое количество суперинтерфейсов, а также суперкласс. Все объекты класса указываются перед объектом с реализованными методами.

Еще одно расширение синтаксиса в Nashorn позволяет определять анонимные подклассы интерфейсов или абстрактных классов. Если после оператора `new` *ОбъектКлассаJava* следует объект JavaScript, то в конечном итоге возвращается объект расширенного класса. Ниже приведен характерный тому пример.

```
var iter = new java.util.Iterator {
  next: function() Math.random(),
  hasNext: function() true
}
```

Если супертип является абстрактным классом и имеет единственный абстрактный метод, то этот метод можно даже и не именовать. Вместо этого функция передается так, как если бы она была параметром конструктора:

```
var task = new java.lang.Runnable(function() { print('Hello') })
    // Задачей является объект анонимного класса,
    // реализующего интерфейс Runnable
```

ВНИМАНИЕ. Данный синтаксис нельзя применять для расширения *конкретного* класса. Например, в операторе **`new java.lang.Thread(function() { print('Hello')`** вызывается конструктор класса **`Thread`** [в данном случае конструктор **`Thread(Runnable)`**]. В результате вызова оператора **`new`** возвращается объект класса **`Thread`**, но не его подкласса.

Если в подклассе требуются переменные экземпляра, они должны быть введены в объект JavaScript. В качестве примера ниже приведен итератор, производящий последовательность из десяти произвольных чисел. Обратите внимание на то, что в

методах next() и hasNext() в JavaScript делается ссылка this.count на переменную экземпляра.

```
var iter = new java.util.Iterator {
    count: 10,
    next: function() { this.count--; return Math.random() },
    hasNext: function() this.count > 0
}
```

При переопределении метода можно вызвать метод из суперкласса, хотя это и кажется чрезмерным. Так, в результате вызова Java.super(obj) получается объект, для которого можно вызвать метод из суперкласса, к которому относится объект obj. Но этот объект должен быть доступным. Ниже показано, как этого добиться.

```
var arr = new (Java.extend(java.util.ArrayList)) {
  add: function(x) {
    print('Adding ' + x);
    return Java.super(arr).add(x)
  }
}
```

При вызове метода arr.add('Fred') выводится сообщение перед вводом значения в списочный массив. Следует заметить, что для вызова метода Java.super(arr) требуется переменная arr, в которой устанавливается значение, возвращаемое оператором new. Вызов Java.super(this) не действует, поскольку в итоге получается только объект JavaScript, определяющий метод, но не заместитель этого объекта в Java. Таким образом, вызов Java.super() оказывается удобным только для определения отдельных объектов, а не подклассов.

НА ЗАМЕТКУ. Вместо вызова **Java.super(arr).add(x)** можно также воспользоваться синтаксисом вызова **arr.super$add(x)**.

14.3.10. Исключения

Если в методе Java генерируется исключение, его можно перехватить в сценарии JavaScript обычным образом:

```
try {
  var first = list.get(0)
  ...
} catch (e) {
  if (e instanceof java.lang.IndexOutOfBoundsException)
    print('list is empty')
}
```

Следует, однако, иметь в виду, что в JavaScript для перехвата исключений употребляется только один оператор catch, в отличие от Java, где исключения можно перехватывать по их типу. И это вполне соответствует принципу динамически типизированных языков, где все запросы типов происходят во время выполнения.

14.4. Написание сценариев командного процессора средствами Nashorn

Если требуется автоматизировать задачи, неоднократно выполняемые на компьютере, то для этой цели обычно составляется *сценарий командного процессора*, воспроизводящий последовательность команд операционной системы. На моем компьютере имеется каталог `~/bin`, заполненный десятками сценариев командного процессора, предназначенных для выгрузки файлов на мой веб-сайт, в мой блог, фотоархив и FTP-сайт моего издателя; преобразования изображений в формат, удобный для их публикации в моем блоге; массовой рассылки сообщений электронной почты моим студентам; резервного копирования данных на моем компьютере один раз в сутки и т.д.

Теперь это сценарии командного процессора **bash**, а раньше, когда я пользовался Windows, это были командные файлы. Что же в них не так? Дело в том, что рано или поздно в таких сценариях возникает потребность в ветвлении и циклах. По каким-то причинам большинство разработчиков командных процессоров плохо разбирались в основах проектирования языков программирования. В частности, переменные, ветвления, циклы и функции реализованы в командном процессоре **bash** из рук вон плохо, а командный язык в Windows в этом отношении еще хуже. У меня имеется несколько сценариев командного процессора **bash**, которые поначалу были скромных размеров, но со временем разрослись настолько, что стали практически неуправляемыми. И это типичный недостаток подобных сценариев.

А почему бы не написать подобные сценарии на Java? А потому, что язык Java слишком многословен. Если вызывать внешние команды с помощью метода `Runtime.exec()`, то придется управлять стандартными потоками ввода-вывода данных и сообщений об ошибках. Поэтому для написания сценариев командного процессора разработчики Nashorn предлагают в качестве альтернативы язык JavaScript. Его синтаксис относительно прост, а интерпретатор Nashorn предоставляет некоторые удобства специально для программирующих на уровне командного процессора.

14.4.1. Выполнение команд из командного процессора

Чтобы воспользоваться в Nashorn расширениями, предназначенными для написания сценариев, достаточно выполнить следующую команду:

```
jjs -scripting
```

После этого команды можно выполнять из командного процессора, заключая их в обратные кавычки, как показано ниже.

```
`ls -al`
```

Данные и сообщения об ошибках при выполнении последней команды выводятся в стандартные потоки и фиксируются в переменных `$OUT` и `$ERR` соответственно. А код завершения команды фиксируется в переменной `$EXIT`. (Нулевым кодом принято обозначать успешное завершение, а ненулевыми кодами — состояния ошибок.)

Чтобы зафиксировать стандартный вывод, достаточно заключить команду в обратные кавычки, а результат ее выполнения присвоить переменной следующим образом:

```
var output = `ls -al`
```

Если для исполняемой команды требуется предоставить стандартный ввод, достаточно воспользоваться следующей командой:

```
$EXEC(команда, ввод)
```

Например, приведенная ниже команда организует передачу команде **grep -v class** результата, выводимого из команды **ls -al**.

```
$EXEC('grep -v class', `ls -al`)
```

Это, конечно, не совсем конвейер. Но его, если требуется, нетрудно реализовать (см. упражнение 10 в конце этой главы).

14.4.2. Интерполяция символьных строк

Выражения в конструкции `${...}` вычисляются в символьных строках, заключенных в двойные и обратные кавычки. Это так называемая *интерполяция символьных строк*. В следующем примере:

```
var cmd = "javac -classpath ${classpath} ${mainclass}.java"
$EXEC(cmd)
```

или просто

```
`javac -classpath ${classpath} ${mainclass}.java`
```

содержимое переменных `classpath` и `mainclass` вставляется в исполняемую команду.

В конструкции `${...}` можно указывать произвольные выражения следующим образом:

```
var message = "The current time is ${java.time.Instant.now()}"
  // устанавливает в сообщении символьную строку, например,
  // "The current time is 2013-10-12T21:48:58.545Z"
  // (Текущее время: 2013-10-12T21:48:58.545Z)
```

Как и в командном процессоре **bash**, интерполяция не действует в символьных строках, заключаемых в одиночные кавычки. Ниже приведен характерный тому пример.

```
var message = 'The current time is ${java.time.Instant.now()}'
    // устанавливает в сообщении следующую символьную строку:
    // "The current time is ${java.time.Instant.now()}"
    // (Текущее время: ${java.time.Instant.now()}"
```

Символьные строки также интерполируются в документах, встраиваемых в сценарии. Такие документы оказываются полезными, когда многие строки читаются по команде из стандартного потока ввода, тогда как автору сценария не требуется вводить

их в отдельный файл. В качестве примера ниже показано, каким образом инструментальное средство администрирования GlassFish снабжается командами.

```
name='myapp'
dir='/opt/apps/myapp'
$EXEC("asadmin", <<END)
start-domain
start-database
deploy ${name} ${dir}
exit
END
```

Конструкция <<END означает следующее: "Вставить символьную строку, начинающуюся со следующей строки и завершающуюся строкой END". (Вместо строки END можно указать любой идентификатор, отсутствующий в символьной строке.)

Следует также иметь в виду, что наименование и местоположение прикладной программы интерполируются. Интерполяция символьных строк и встраиваемые документы доступны только в режиме сценариев.

14.4.3. Ввод данных в сценарий

Сценарий можно снабдить аргументами командной строки. В строке команды **jjs** можно ввести несколько файлов сценариев, и поэтому эти файлы нужно отделить от аргументов разделителем --, как показано ниже.

```
jjs script1.js script2.js -- arg1 arg2 arg3
```

В файле сценария аргументы командной строки вводятся в массив arguments следующим образом:

```
var deployCommand = "deploy ${arguments[0]} ${arguments[1]}"
```

Вместо массива arguments в команде **jjs** можно использовать переменную $ARG. Если эта переменная используется вместе с интерполяцией символьных строк, то для указания аргументов потребуются два знака **$**, как показано ниже.

```
var deployCommand = "deploy ${$ARG[0]} ${$ARG[1]}"
```

С помощью объекта ENV в сценарии можно также получить доступ к переменным окружения командного процессора следующим образом:

```
var javaHome = $ENV.JAVA_HOME
```

Используя функцию readLine(), в режиме сценариев можно предоставить пользователю подсказку для ввода данных:

```
var username = readLine('Username: ')
```

И наконец, для завершения сценария служит функция exit(). Эту функцию можно снабдить дополнительным кодом завершения следующим образом:

```
if (username.length == 0) exit(1)
```

Первая строка сценария может начинаться с последовательности символов **#!**, после которой следует местоположение интерпретатора сценария, как показано ниже.

```
#!/opt/java/bin/jjs
```

В Linux, Unix и Mac OS X файл сценария можно сделать исполняемым, введя каталог сценариев в переменную окружения `PATH`, а затем выполнив этот файл в виде сценария `script.js`. Если сценарий начинается с последовательности символов **#!**, то режим сценариев активизируется автоматически.

ВНИМАНИЕ. Если последовательность символов **#!** употребляется в сценарии вместе с аргументами командной строки, то пользователям сценария придется ввести разделитель **--** перед аргументами следующим образом:

```
script.js -- arg1 arg2 arg3
```

Упражнения

1. В технологии JavaServer Pages веб-страница состоит из HTML-разметки и кода Java, как показано в следующем примере:

   ```
   <ul>
   <% for (int i = 10; i >= 0; i--) { %>
     <li><%= i %></li>
   <% } %>
   <p>Liftoff!</p>
   ```

   Все, что находится за пределами разметки `<%...%>` и `<%=...%>`, выводится в исходном виде, а код в пределах этой разметки вычисляется. Если разметка начинается с разделителя `<%=`, то результат вычисления добавляется к тому, что выводится. Реализуйте программу, читающую такую веб-страницу, превращающую ее в метод Java, выполняющую его и получающую результирующую страницу.

2. Организуйте вызов из программы на Java метода `JSON.parse()` в JavaScript, чтобы преобразовать символьную строку из формата JSON в объект JavaScript, а затем обратно в символьную строку. Сделайте это, во-первых, с помощью метода `eval()`; во-вторых, с помощью метода `invokeMethod()`; и в-третьих, вызвав метод Java через следующий интерфейс:

   ```
   public interface JSON {
     Object parse(String str);
     String stringify(Object obj);
   }
   ```

3. Насколько целесообразна компиляция средствами Nashorn? Напишите сценарий JavaScript для сортировки массива простейшим способом перестановки до тех пор, пока элементы массива не будут отсортированы. Сравните время выполнения компилируемой и интерпретируемой версий этого сценария. В качестве примера ниже приведена функция JavaScript для вычисления очередной перестановки.

```
function nextPermutation(a) {
  // найти наибольший невозрастающий суффикс,
  // начиная с элемента массива a[i]
  var i = a.length - 1
  while (i > 0 && a[i - 1] >= a[i]) i--
    if (i > 0) {
      // поменять местами элемент массива a[i - 1] и крайний справа
      // элемент a[k] > a[i - 1], учитывая, что a[i] > a[i - 1]
      var k = a.length - 1
      while (a[k] <= a[i - 1]) k--
      swap(a, i - 1, k)
    } // Иначе суффикс представляет весь массив

    // обратить суффикс
    var j = a.length - 1
    while (i < j) { swap(a, i, j); i++; j-- }
}
```

4. Найдите реализацию языка Scheme, совместимую с прикладным программным интерфейсом API для сценариев. Напишите функцию вычисления факториала на Scheme и организуйте ее вызов из кода Java.

5. Выберите для исследования какую-нибудь часть прикладного программного интерфейса Java API, например класс `ZonedDateTime`. Проведите с помощью утилиты **`jjs`** в режиме командной строки эксперименты с построением объектов и вызовом методов, наблюдая возвращаемые значения. Насколько это проще, чем писать тестовые программы на Java?

6. Выполните утилиту **`jjs`** и воспользуйтесь библиотекой потоков данных, чтобы выработать решение следующей задачи: вывести из файла все однозначные длинные слова (больше 12 букв) в отсортированном порядке. Сначала организуйте чтение слов, а затем фильтрацию длинных слов и т.д. Сравните этот диалоговый способ программирования с привычным для вас способом.

7. Выполните утилиту **`jjs`** и сделайте следующий вызов:

   ```
   var b = new java.math.BigInteger('1234567890987654321')
   ```

   Затем отобразите содержимое переменной `b`, просто введя **`b`** и нажав клавишу <Enter>. Что вы получите в итоге? Какое значение возвращается в результате вызова `b.mod(java.math.BigInteger.TEN)`? Почему значение переменной `b` отображается столь необычно? Как отобразить фактическое значение переменной `b`?

8. В конце раздела 14.3.9 было показано, каким образом можно расширить класс `ArrayList`, чтобы протоколировать каждый вызов метода `add()`. Но это подходит только для одного объекта. Напишите функцию JavaScript, которая должна служить фабрикой для таких объектов, позволяя формировать любое количество списочных массивов для протоколирования.

9. Напишите функцию JavaScript `pipe()`, принимающую в качестве параметров последовательность команд из командного процессора, направляющую вывод из предыдущей команды по конвейеру на вход следующей команды и возвращающей результат, выводимый последней командой, как, например,

`pipe('find .', 'grep -v class', 'sort')`. Для этого просто выполняйте повторно команду `$EXEC`.

10. Решение, предложенное в предыдущем упражнении, не столь совершенно, как настоящий конвейер в Unix, поскольку выполнение следующей команды начинается только по завершении предыдущей. В качестве выхода из этого положения воспользуйтесь классом `ProcessBuilder`.
11. Напишите сценарий, выводящий значения всех переменных окружения.
12. Напишите сценарий `nextYear.js` для получения возраста пользователя и вывода результата `"Next year, you will be ..."` (В следующем году вам будет ...) с прибавлением `1` к введенной величине возраста пользователя. Возраст может быть указан в режиме командной строки или в переменной окружения `AGE`. Если же он так или иначе отсутствует, то предложите пользователю ввести его.

Предметный указатель

З

И

К

Л

М

Н

О

П

Т

У

Ф

Ц

Я

Java SE 8
Вводный курс

Кей С. Хорстманн

www.williamspublishing.com

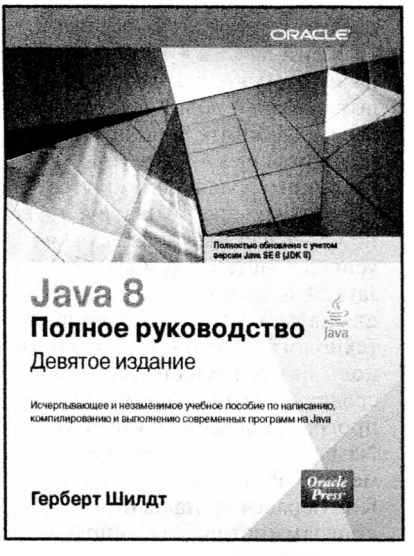
В этом кратком руководстве рассматриваются нововведения в версии Java SE 8, главными из которых являются лямбда-выражения и новые потоки ввода-вывода. В книге поясняются на практических примерах исходного кода и снабжаются полезными советами эти и другие не менее важные новшества и усовершенствования в версии Java SE 8, в том числе библиотека для даты, времени и календаря, технология JavaFX для разработки пользовательских интерфейсов, средства параллельного программирования, интерпретатор Nashorn языка JavaScript и прочие мелкие изменения.
Книга рассчитана на опытных программистов, стремящихся писать в недалекой перспективе надежный, эффективный и безопасный код на Java.

ISBN 978-5-8459-1900-7 **в продаже**

JAVA
ПОЛНОЕ РУКОВОДСТВО

ДЕВЯТОЕ ИЗДАНИЕ

Герберт Шилдт



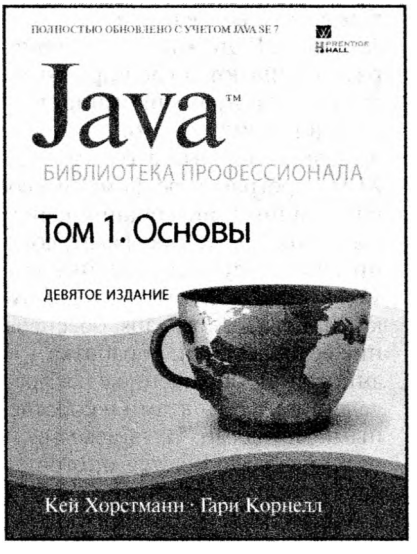
www.williamspublishing.com

Эта книга является исчерпывающим справочным пособием по языку программирования Java, обновленным с учетом последней версии Java SE 8. В удобной и легко доступной для изучения форме в ней подробно рассматриваются все языковые средства Java, в том числе синтаксис, ключевые слова, операции, управляющие и условные операторы, элементы объектно-ориентированного программирования (классы, объекты, методы, обобщения, интерфейсы, пакеты, коллекции), аплеты и сервлеты, библиотеки классов наряду с такими нововведениями, как стандартные интерфейсные методы, лямбда-выражения, библиотека потоков ввода-вывода, технология JavaFX. Основные принципы и методики программирования на Java представлены в книге на многочисленных и наглядных примерах написания программ. Книга рассчитана на широкий круг читателей, интересующихся программированием на Java.

ISBN 978-5-8459-1918-2 **в продаже**

JAVA™
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТОМ 1. ОСНОВЫ
ДЕВЯТОЕ ИЗДАНИЕ

КЕЙ ХОРСТМАНН
ГАРИ КОРНЕЛЛ



www.williamspublishing.com

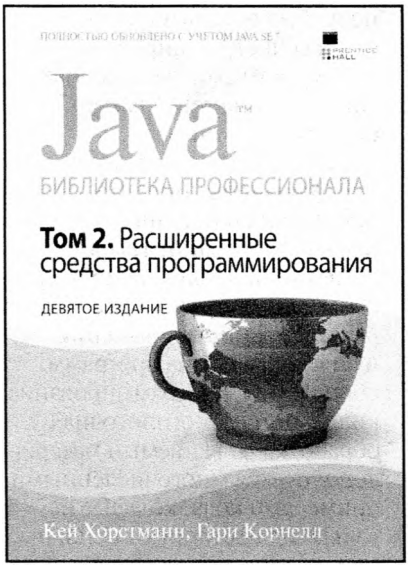
Это первый том обновленного, девятого издания исчерпывающего справочного руководства по программированию на Java с учетом всех нововведений версии Java SE 7. В этом томе подробно рассматриваются основы программирования на Java, в том числе основные типы и фундаментальные структуры данных, объектно-ориентированное программирование и его реализация в Java, интерфейсы, программирование графики средствами библиотеки Swing, обработка событий и исключений, развертывание приложений и аплетов, отладка программ, обобщенное программирование, коллекции и многопоточная обработка. Излагаемый материал дополняется многочисленными примерами кода, которые не только иллюстрируют основные понятия, но и демонстрируют практические приемы программирования на Java. Книга рассчитана на программистов разной квалификации и будет также полезна студентам и преподавателям дисциплин, связанных с программированием на Java.

ISBN 978-5-8459-1869-7

в продаже

JAVA™. БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТОМ 2. РАСШИРЕННЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ
Девятое издание

Кей Хорстманн,
Гари Корнелл



www.williamspublishing.com

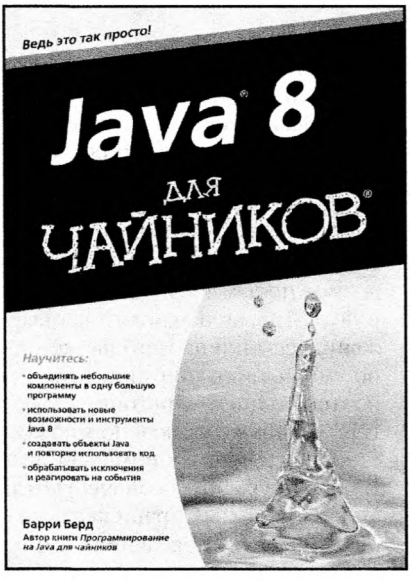
Это второй том обновленного, девятого издания исчерпывающего справочного руководства по программированию на Java с учетом всех нововведений в версии Java SE 7. В этом томе подробно рассматриваются расширенные средства программирования на Java, в том числе потоки ввода-вывода, файловый ввод-вывод, XML, программирование сетевое и баз данных, интернационализация прикладных программ, дополнительные функциональные возможности библиотек Swing и AWT, компоненты JavaBeans, обеспечение безопасности, обработка аннотаций, обращение с распределенными объектами и собственными методами. Излагаемый материал дополняется многочисленными примерами кода, которые не только иллюстрируют поясняемые понятия, но и демонстрируют практические приемы усовершенствованного программирования на Java.

Книга рассчитана на программистов разной квалификации и будет также полезна студентам и преподавателям дисциплин, связанных с программированием на Java.

ISBN 978-5-8459-1870-3 **в продаже**

Java 8 для чайников

Барри Берд



www.dialektika.com

Java — современный объектно-ориентированный язык программирования. Программа, написанная на Java, способна выполняться практически на любом компьютере. Зная Java, можно создавать мощные мультимедийные приложения для любой платформы. Десятки тысяч программистов начинали изучать Java с помощью предыдущих изданий этой книги. Теперь ваша очередь! Независимо от того, на каком языке вы программировали раньше (и даже если вы никогда прежде не программировали), вы быстро научитесь создавать современные кроссплатформенные приложения, используя возможности Java 8.

Основные темы книги:

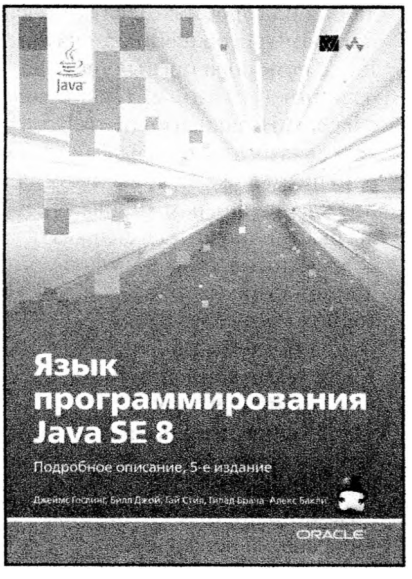
- ключевые концепции Java;
- грамматика языка;
- повторное использование кода;
- циклы и условные конструкции;
- принципы объектно-ориентированного программирования;
- обработка исключений;
- создание апплетов Java;
- как избежать распространенных ошибок.

ISBN 978-5-8459-1928-1 **в продаже**

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA SE 8

Подробное описание, 5-е издание

Джеймс Гослинг,
Билл Джой,
Гай Стил,
Гилад Брача,
Алекс Бакли



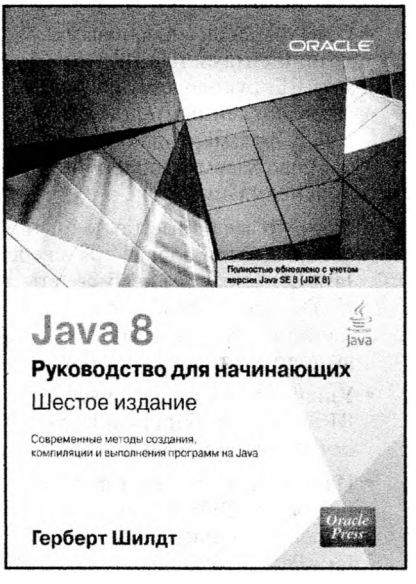
Книга написана разработчиками языка Java и является полным техническим справочником по этому языку программирования. Она обеспечивает полный, точный и подробный охват всех аспектов языка программирования Java. В ней полностью описаны новые возможности, добавленные в Java SE 8, включая лямбда-выражения, ссылки на методы, методы по умолчанию, аннотации типов и повторяющиеся аннотации. В книгу также включено множество поясняющих примечаний. В ней аккуратно обозначены отличия формальных правила языка от практического поведения компиляторов.

www.williamspublishing.com

ISBN 978-5-8459-1875-8 **в продаже**

Java 8: руководство для начинающих *6-е издание*

Герберт Шилдт



www.williamspublishing.com

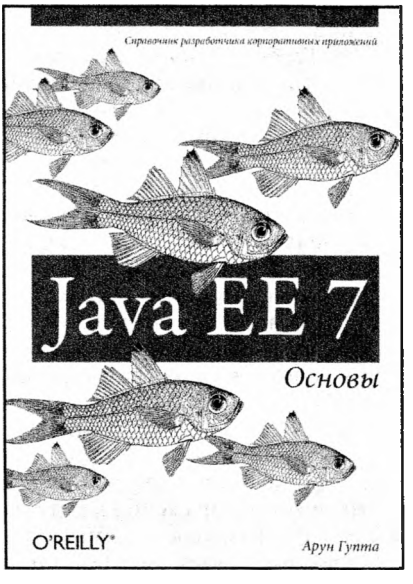
Настоящее, 6-е, издание бестселлера, обновленное с учетом всех новинок последнего выпуска Java Platform, Standard Edition 8 (Java SE 8), позволит новичкам сразу же приступить к программированию на языке Java. Герберт Шилдт, всемирно известный автор множества книг по программированию, уже в начале книги знакомит читателей с тем, как создаются, компилируются и выполняются программы, написанные на языке Java. Далее объясняются ключевые слова, синтаксис и языковые конструкции, образующие ядро Java. Кроме того, в книге рассмотрены темы повышенной сложности: многопоточное программирование, обобщенные типы и средства библиотеки Swing. Не остались без внимания автора и такие новейшие возможности Java SE 8, как лямбда-выражения и методы интерфейсов, используемые по умолчанию. В заключение автор знакомит читателей с JavaFX — новой перспективной технологией создания современных графических интерфейсов пользователя, отличающихся изящным внешним видом и богатым набором элементов управления.

ISBN 978-5-8459-1955-7

в продаже

Java EE 7
Основы

Арун Гупта



www.williamspublishing.com

Книга написана одним из ведущих разработчиков проекта Java EE, и каждая глава в ней посвящена рассмотрению одной из ключевых спецификаций платформы, включая WebSockets, Batch Processing, RESTful Web Services и Java Message Service.

- Ознакомьтесь с ключевыми компонентами платформы Java EE, руководствуясь многочисленными примерами в виде фрагментов кода, сопровождаемых подробными пояснениями автора.
- Изучите все новые технологии, которые были добавлены в версии Java EE 7, включая веб-сокеты, JSON, пакетную обработку и утилиты параллельного выполнения.
- Узнайте о применении веб-служб RESTful, служб на основе SOAP и службы сообщений Java (JMS).
- Изучите технологии Enterprise JavaBeans, CDI (Contexts and Dependency Injection) и Java Persistence.
- Узнайте о том, каким изменениям подверглись различные компоненты при переходе от Java EE 6 к Java EE 7.

ISBN 978-5-8459-1896-3 **в продаже**

Java SE 8 Базовый курс

В версии Java SE 8 внедрены значительные усовершенствования, оказывающие влияние на технологии и прикладные программные интерфейсы API, образующие ядро платформы Java. Многие из прежних принципов и приемов программирования на Java больше не нужны, а новые средства вроде лямбда-выражений повышают производительность труда программистов, хотя разобраться в этих нововведениях не так-то просто. Эта книга является полным, хотя и кратким справочником по версии Java SE 8. Она написана Кеем С. Хорстманном, автором книги *Java SE 8. Вводный курс* и классического двухтомного справочника по предыдущим версиям Java, и служит незаменимым учебным пособием для быстрого и легкого изучения этого языка и его библиотек. Учитывая масштабы Java и разнообразие новых языковых средств, внедренных в версии Java SE 8, материал этой книги подается небольшими порциями для быстроты усвоения и простоты понимания. Многочисленные практические рекомендации автора книги и примеры кода помогут читателям, имеющим опыт программирования на Java, быстро воспользоваться преимуществами лямбда-выражений, потоков данных и прочими усовершенствованиями языка и платформы Java. В книге освещается все, что нужно знать прикладным программистам о современной версии Java, включая следующее.

- Ясное и доходчивое изложение синтаксиса лямбда-выражений, позволяющих лаконично выражать выполняемые действия
- Подробное введение в новый прикладной программный интерфейс API потоков данных, благодаря которому обработка данных становится более гибкой и эффективной
- Рассмотрение основных принципов параллельного программирования, стимулирующих к разработке программ с точки зрения взаимодействия параллельно выполняемых задач, а не низкоуровневых потоков исполнения и блокировок
- Современный взгляд на новые библиотеки вроде даты и времени
- Обсуждение других новых средств, которые могут быть особенно полезны для разработчиков серверных и мобильных приложений

Это книга станет неоценимым источником информации для всех, кто стремится писать в недалекой перспективе самый надежный, эффективный и безопасный код на Java: как начинающих, так и опытных программистов.

Кей С. Хорстманн – автор книги *Scala for the Impatient* (издательство Addison-Wesley, 2012 г.), *Java SE 8. Вводный курс* (ИД "Вильямс", 2014 г.) и основной автор двухтомного издания *Java. Библиотека профессионала* (ИД "Вильямс", 2014 г.). Он также написал десяток других книг для профессиональных программистов и изучающих вычислительную технику. Кей занимает должность профессора на кафедре вычислительной техники при университете штата Калифорния в Сан-Хосе и является обладателем почетного звания "Чемпион по Java".

Категория: программирование
Предмет рассмотрения: язык Java, версия SE 8
Уровень: для начинающих и пользователей средней квалификации

ISBN 978-5-8459-2004-1

ВИЛЬЯМС
www.williamspublishing.com
horstmann.com/java8
Addison-Wesley
ALWAYS LEARNING
PEARSON